Elmar Langetepe

Online Motion Planning

MA INF 1314

Contents

List of Figures

List of Algorithms

Introduction

This lecture considers tasks for autonomous agents. In general, constructing autonomous machines is a very complex challenge and has many different engineering and scientific aspects, some of which are given in the following list.

- Elektronic devices

- Mechanical devices

- Control/Process engineering

- Artificial Intelligence

- Softwareengineering

- $\vdots$

- Plans: Algorithmic/Motion planning

- Full information (offline)/ Incomplete information (online)

- Input: Geometry of the Environment

As part of the algorithm track of the master program we will concentrate on the item *Algorithms*. That is, we concentrate on the description and analysis of efficient schedules for solving motion planning tasks for autonomous agents. Besides, we concentrate on problem definitions and models that take the geometry of the scene into account. In this sense the scientific aspects of this course are part of the scientific area called Computational Geometry. Furthermore we consider online problems, which means that the full information of the problem is not given in advance. The agent has to *move* around and collects more information.

We will mainly concentrate on the ground tasks of autonomous agents in unknown environments such as

- Searching for a goal,

- Exploration of an environment,

- Escaping from a labyrinth,

and we consider different abilities of the agents some of which are

- Continuous/discrete vision,

- Touch sensor/compass,

- Building a map/constant memory.

The first concern is that we construct correct algorithms which always fulfil the task. Second we concentrate on the efficiency of the corresponding strategy. We would like to analyse performance guarantees and would like to provide for formal proofs. The course is related to the undergraduate course on *Offline motion planning*. In the offline case the information for the task is fully given and we only have to compute the best path for the agent. The offline solution will be used as a comparison measure for the online case. This is a well known concept for online problems in general.

Chapter 1

Labyrinths, grids and graphs

In this section we first concentrate on discrete environments based on grid structures. For the grid structure we consider an agent that can move from one cell to a neighbouring cell with unit cost. We start with the task of searching for a goal in a very special grid environment. After that we ask for visiting all cells, which means that we would like to explore the environment. For this task the grid environment is only partially known, by a touch sensor the agent can only detect the neighbouring cells. The agent can build a map. Exploration and Searching are closely related. If we are searching for an unknown goal, it is clear that in the worst-case the whole environment has to be explored. The main difference is the performance of these *online* tasks. As a comparison measure we compare the length of the agent's path to the length of the optimal path under full information. Thus, in the case of searching for a goal, the comparison measure is the shortest path to the goal.

At the end of the section we turn over to the exploration task in general graphs under different additional conditions.

1.1   Shannons Mouse Algorithm

Historically the first online motion planning algorithm for an autonomous agent was designed by Claude Shannon [Sha52, Sha93] in 1950. He considered a 5×5 cellular labyrinth, the inner walls of the labyrinth could be placed around arbitrary cells. In principle, he constructed a labyrinth based on a grid environment; see Figure 1.1.

The task of his electronical mouse was to find a target, i.e. the cheese, located on one of the fields of the grid. The target and the start of the mouse were located in the same *connected component* of the *grid labyrinth*. The electronical mouse was able to move from one cell to a neighbouring cell. Additionally, it could (electronically) mark any cell by a label N, E, S, W which indicates in which direction the mouse left the cell at the last visit. This label is updated after leaving the cell. With theses abilities the following algorithm was designed.

Algorithm 1.1 Shannons Maus

- Initialize any cell by the label N for 'North'.

- While the goal has not been found:
  starting from the label direction, search for the first cell in clockwise order that can be visited. Change the label to the corresponding direction and move to this neighbouring cell.

Sutherland [Sut69] has shown that:

Theorem 1.1   *Shannon's Algorithms (Algorithm 1.1) is correct. For any labyrinth, any starting and any goal the agent will find the goal, if a path from the start to the goal exists.*

Figure 1.1: Shannons original mouse labyrinth.
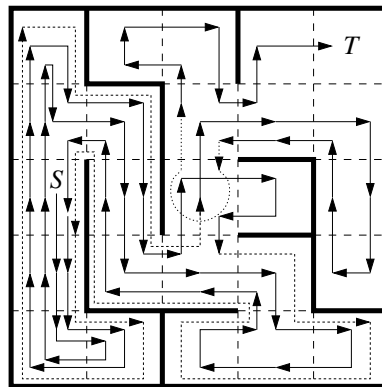
Figure 1.2: An example of the execution of Shannons Algorithm.

Proof. We omit the goal and show that any cell in the connected component of the start will be visited infinitely often.   □

Exercise 1 *Formalize the above proof sketch!*

As shown in Figure 1.2 the path of Shannons Mouse is not very efficient.

1.2   Intuitive connection of labyrinths, grids and graphs

For a human a labyrinth consists of corridors and connection points. In this sense the environment for Shannons task can be considered to be a labyrinth. Obviously any such labyrinth can be modeled by a planar graph.[1] More precisely the environment for Shannons task is a grid graph. Figure 1.3 shows the corresponding intuitive interpretations.

For any intuitive labyrinth there is a labyrinth-graph. On the other hand for any planar graph we can build some sort of labyrinth. This is not true for general graphs. For example the complete graph K_5 has no planar representation and therefore a correspondance to a labyrinth does not exist.

1.3   A lower bound for online graph exploration

We consider the following model. Assume that a graph $G = (V, E)$ is given. If the agent is located on a vertex it detects all neighbouring vertices. Let us assume that moving along an edge can be done with

---

[1]A graph, that has an intersection free representation in the plane.

(i) Labyrinth   (ii) Labyrinthgraph   (iii) Gittergraph
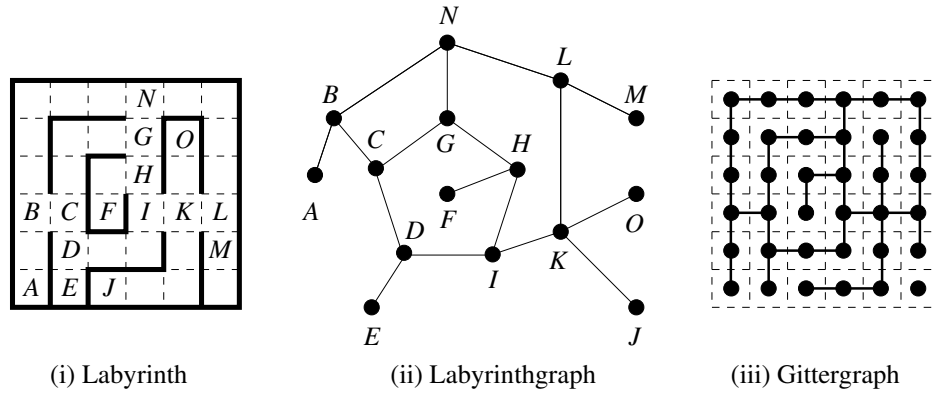
Figure 1.3: Labyrinth, labyrinth-graph and gridgraph.

unit cost. The task is to visit all edges and vertices and return to the start. The agent has the ability of building a map. If we apply a DFS (depth first search) for the edges we will move along any edges twice. DFS can run online. The best offline strategy has to visit any edge at least once. In this sense DFS is a 2-approximation.

The comparison and approximation between online and offline is represented by the following concept. A strategy that runs under incomplete information is denoted as an **Online–Strategy**. On the other hand an **Offline–Strategy** solves the same task with full information. In the above example the offline strategy is the shortest round trip that visits all edges of the graph.

The performance measure for Online-Algorithms is the so-called *competitive ratio*.

Definition 1.2   (Sleator, Tarjan, 1995)
Let Π be a problem class and S be a strategy, that solves any instance $P \in \Pi$.
Let $K_S(P)$ be the cost of S for solving P.
Let $K_{opt}(P)$ be the cost of the optimal solution for P.
The strategy S is denoted to be *c–competitive*, if there are fixed constants $c, \alpha > 0$, so that for all $P \in \Pi$

$$K_S(P) \le c \cdot K_{opt}(P) + \alpha$$

holds.

The additive constant α is often used for starting situations. For example if we are searching for a goal and have only two unknown options, the goal might be very close to the start, the unsuccessful step will lead to an arbitrarily large competitive ratio. This is not intended. Sometimes we can omit the additive constant, if we have additional assumptions. For example we can assume that the goal is at least distance 1 away from the start.

As already mentioned DFS on the edges visits any edge at most twice. There are graphs where the optimal offline solution also has to visit any edge twice. For such examples DFS is optimal with ratio 1. Now we are searching for a lower bound for the competitive ratio. That is, we would like to construct example such that any possible online strategy fails within a ratio of 2.

Theorem 1.3   *(Icking, Kamphans, Klein, Langetepe, 2000)*
For the online-exploration of a graph $G = (V, E)$ for visiting all edges and vertices of G there is always an arbitrarily large example such that any online strategy visits roughly twice as much edges in comparison to the optimal offline strategy. DFS always visit no more than twice as much edges against the optimum.
[IKKL00a]

Proof. The second part is clear because DFS visits exactly any edge twice. Any optimal strategy has to visit at least the edges.

The robot should explore a gridgraph and starts in a vertex s. Finally, the agent has to return to s. We construct an *open* corridor and offer two directions for the agent. At some moment in time the agent has explored ℓ new vertices in the corridor. If this happens we let construct a conjunction at one end s' of the corridor. At this bifurcation two open corridors are build up which *run* back into the direction of s. If the agent proceeds one of the following events will happen.

1. The agent goes back to s.

2. The agent has visited more than $\ell + 1$ edges in one of the new corridors.

Let ℓ_1 denote the length of the part of the starting open corridor into the opposite direction of s'. Let ℓ_2 and ℓ_3 denote the length of the second and third open corridor.

We analyse the edge visits $|S_{ROB}|$ that an arbitrary strategy S_{ROB} has done so far.

1. $|S_{ROB}| \geq 2\ell_1 + (\ell - \ell_1) + 2\ell_2 + 2\ell_3 + (\ell - \ell_1) = 2(\ell + \ell_2 + \ell_3)$, see Figure 1.4. Now we close the corridors at the open ends. From now on the agent still requires $|S_{OPT}| = 2(\ell + \ell_2 + \ell_3) + 6$ edge visits, where S_{OPT} is the optimal strategy if the situation was known from the beginning. Thus we have: $|S_{ROB}| \geq 2|S_{OPT}| - 6$.
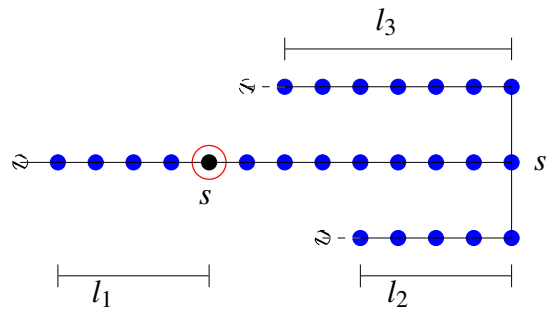
Figure 1.4: The agent return to s.

2. W.l.o.g. the agent has explored $\ell + 1$-ten vertices in corridor 3. We have $|S_{ROB}| \geq 2\ell_1 + (\ell - \ell_1) + 2\ell_2 + (\ell + 1)$. We connect corridor 3 with corridor 1(see Figure 1.5) and close corridor 2. The agent still requires $\ell + 1 + 2(\ell_2 + 1) + (\ell - \ell_1)$ edge visits; in total at least $4\ell + 4\ell_2 + 4 = 4(\ell + \ell_2) + 4$ edge visits. From $|S_{OPT}| = 2(\ell + 1) + 2(\ell_2 + 1) = 2(\ell + \ell_2) + 4$ we conclude $|S_{ROB}| \geq 2|S_{OPT}| - 4 > 2|S_{OPT}| - 6$.
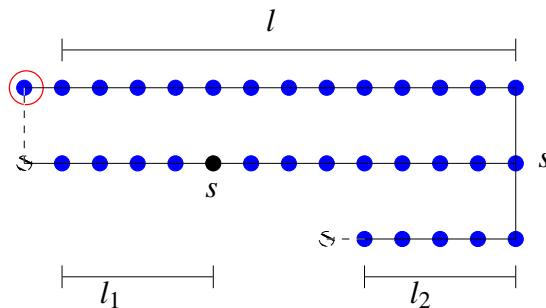
Figure 1.5: The agent has visited $\ell + 1$ vertices in corridor 3.

We have $|S_{ROB}|/|S_{OPT}| \geq 2 - 6/|S_{OPT}|$. We also have $|S_{OPT}| \geq 2(\ell + 1)$ and conclude $2 - 6/|S_{OPT}| > 2 - 6/2\ell = 2 - 3/\ell$. For arbitrary $\delta > 0$ we choose $\ell = \lceil 3/\delta \rceil$ and conclude $|S_{ROB}|/|S_{OPT}| > 2 - \delta$. $\square$

Remark 1.4   *There are always examples so that the optimal exploration tour visits any edge twice.*

Corollary 1.5   *DFS for the Online-Edge-Exploration of general graphs is 2–competitive and optimal.*

Exercise 2   *Show that the same competitive ratio holds, if the return to the starting point is not required.*

Exercise 3   *Consider the problem of exploring the vertices (not the edges) of a graph. If the agent is located at a vertex it detects the outgoing edges but along non-visited edges it is not clear which vertex lies on the opposite side. Does DFS applied on the vertices result in a 2-approximation?*

1.4   Exploration of grid environments

Next we consider a simple discrete grid model. The agent runs inside a grid-environment. In contrast to Shannons the inner obstacles consist of full cells instead of single blocked edges.

We would like to design efficient strategies for such grid environments. First, we give a formal definition.

Definition 1.6

- A **cell** c is a tupel $(x,y) \in \mathbb{N}^2$.

- Two cells $c_1 = (x_1,y_1), c_2 = (x_2,y_2)$ are **adjacent**, if $:\Leftrightarrow |x_1 - x_2| + |y_1 - y_2| = 1$. For a single cell c, exact 4 cells are adjacent.

- Two cells $c_1 = (x_1,y_1), c_2 = (x_2,y_2)$, $c_1 \neq c_2$ are **diagonally adjacent**, if $:\Leftrightarrow |x_1 - x_2| \leq 1 \wedge |y_1 - y_2| \leq 1$. For a single cell c, exact 8 cells are diagonally adjacent.

- A **path** $\pi(s,t)$ from cell s to cell t is a sequence of cells $s = c_1, \ldots, c_n = t$ such that c_i and c_{i+1} are adjacent for $i = 1, \ldots, n-1$.

- A **gridpolygon** P is a set of path-connected cells, i.e., $\forall c_i, c_j \in P : \exists$ path $\pi(c_i, c_j)$, such that $\pi(c_i, c_j) \in P$ verl"auft.

The agent is equipped with a touch sensor so that the agent scans the adjacent cells and their nature (free cell or boundary cell) from its current position. Additionally, the agent has the capability of building a map. The task is to visit all cells of the gridpolygon and return to the start. This problem is NP-hart for known environments; see [IPS82]. We are looking for an efficient Online-Strategy. The agent can move within one step to an adjacent cell. For simplicity we count the number of movements.

The task is related to vacuum-cleaning or lawn-mowing. A cell represents the size of the tool, the tool should visit all cells of the environment. A general polygonal environment P can be approximated by a grid-polygon.
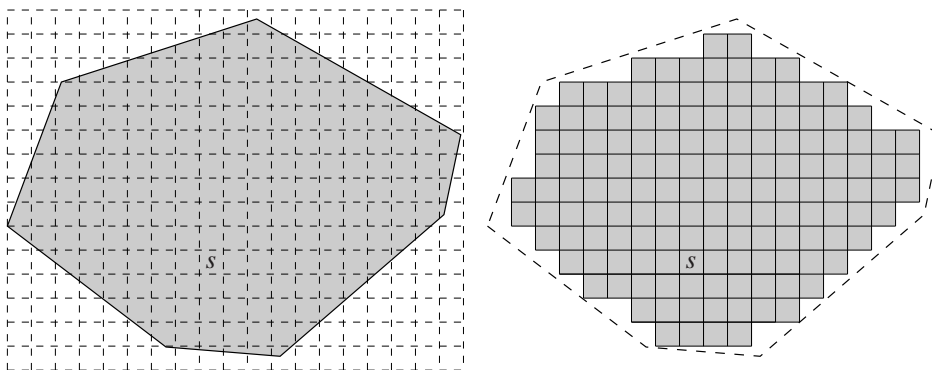
Figure 1.6: A polygon P and the gridpolygon $P_\square$ as a reasonable approximation.

The starting position and orientation of the tool fixes the grid and all connected cells which are entirely inside P belong to the approximation $P_\square$; see Figure 1.6. For any gridpolygon P' we use the following notation. Cells that do not belong to P' but are diagonally adjacent to a cell in P' are called boundary cells. The common edges of the boundary cells and cells of P' are the boundary edges. Let $E(P')$ denote the number of boundary cells or E for short, if the context is clear. The number of cells is denoted by $C(P')$ or C respectively.

From Theorem 1.3 we can already conlcude a lower bound of 2 for the competitive ratio of this problem. On the other hand DFS on the cells finishes the task in $2C - 2$ steps
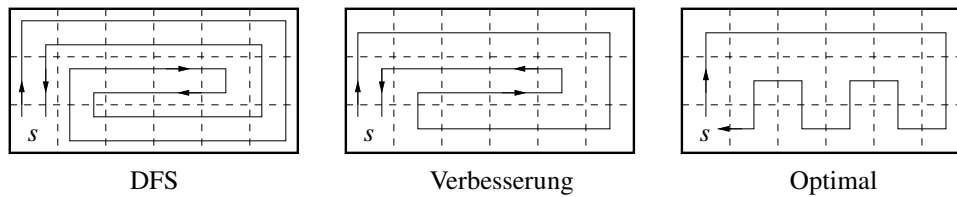
Figure 1.7: Ist DFS optimal?

Exercise 4 *Give a formal proof that for a gridpolygon P the DFS strategy on the cells requires exactly* $2C - 2$ *steps for the exploration (with return to the start) of P.*

But is DFS really the best strategy in general? For fleshy environments DFS obviously is not very efficient. Besides the lower bound construction makes use of corridors only. Compare Figure 1.7: After DFS has visited the *right* neighbour of *s* the environment is fully known and we can improve the strategy. It seems that even the optimal solution could be found in an online fashion in this example. On the other hand there are always *skinny* corridor-like environments where DFS is the best online strategy. Altogether, we require a case sensitive measure for the performance of an online strategy that relies on the existence of large areas. The existence of large *fleshy* areas depends on the relationship between the number of cells C and the number of (boundary) edges E. In Figure 1.7 the environment has 18 edges and 18 cells. In corridor-like environments we have $\frac{1}{2}E \approx C$ in fleshy environments we have $\frac{1}{2}E \ll C$; see also Figure 1.8.
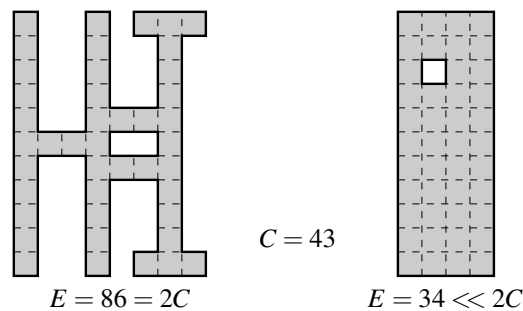
Figure 1.8: The number of boundary edges E in comparison to the number of cells C is a measure for the existence of *fleshy* or *skinny* parts.

1.4.1 Exploration of simple gridpolygons

We first consider *simple* gridpolygons P which do not have any *inner* boundary cell, i.e., also the set of all cells that do not belong to P are path connected.

Note that the lower bound of 2 is not given, because the lower bound construction in the previous section requires the existence of inner obstacles. We make use of a different construction.

Theorem 1.7 *Any online strategy for the exploration (with return to the start) of a simple gridpolygon P of C cells, requires at least $\frac{7}{6}C$ steps for fulfilling the task.*

Proof. We let the agent start in a corner as depicted in Figure 1.9(i) and successively extend the walls. Assume that the agent decides to move to the east first. By symmetry we apply the same arguments, if the agent moves to the south. For the second step the agent has two possibilities (moving backwards can be ignored). Either the strategy leaves the wall by a step to the south (seeFigure 1.9(ii)) or the strategy follows the wall to the east (see Figure 1.9(iii)).

In the first case we close the polygon as shown in Figure 1.9(iv). For this small example the agent requires 8 steps whereas the optimal solution requires only 6 steps which gives a ratio of $\frac{8}{6} \approx 1.3$.
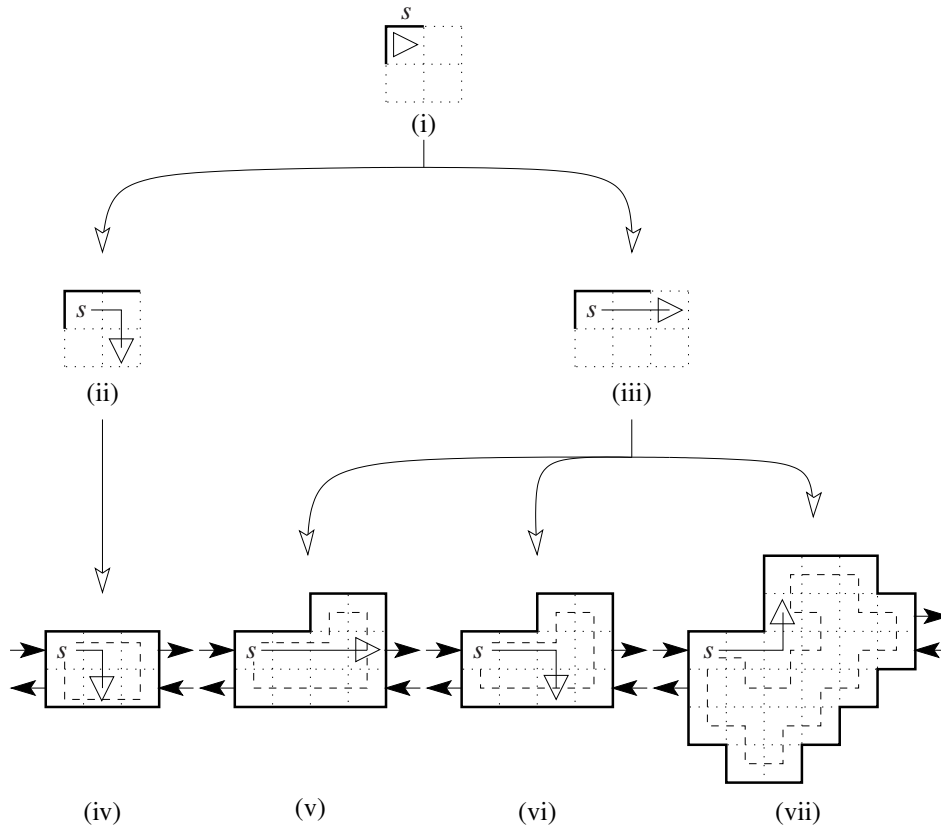
Figure 1.9: A lower bound construction for the exploration of simple gridpolygons.

In the second case we proceed as follows: If the robot leaves the wall (the wall runs upwards), we close the polygon as depicted in Figure 1.9(v) or (vi), respectively. In this small example the agent requires 12, respectively, whereas 10 steps are sufficient.

In the last and most interesting case the agent follows the wall upwards and we present the sophisticated polygon of Figure 1.9(vii). In the offline case an agent requires 24 steps. The online agent already made a mistake and can only finish the task within 24 steps. This can be shown by a tedious case distinction of all further movements. We made use of an implementation that simply checks all possibilities for the next 24 steps. There was no such path that finishes the task. For all cases we guarantee have a worst-case ratio of $\frac{28}{24} = \frac{7}{6} \approx 1.16$.

We use this scheme in order to present a lower bound construction of arbitrary size. Any block has an entrance and exit cell which are marked by corresponding arrows; see Figure 1.9(iv)–(vii). If an agent moves inside the next block, the game starts again. Since the arrows only point in east or west direction we take care that the concatenated construction results in a simple gridpolygon of arbitrary size. as required.   □

Note that the arbitrary-size condition in the above proof is necessary. Assume that we can only construct such examples of fixed size D. This will not result in a lower bound on the competitive ratio. Any reasonable algorithm will explore the fixed envirnment with komeptitive ratio 1 since $\alpha \gg D$ exists, with $|S_{\text{ALG}}| \leq |S_{\text{OPT}}| + \alpha$.

We consider the exploration of a simple gridpolygon by DFS and formalize the strategy; see Algorithm 1.2. The agent explores the polygon by the "Left-Hand-Rule", i.e. the DFS preference is Left before Straight-On before Right. The current direction (North, West, East or South) is stored in the variable dir. The functions cw(dir), ccw(dir) und reverse(dir) result in the corresponding directions of a rotation by 90° in clockwise or counter-clockwise order or by a rotation of 180°, respectively. The predicate unexplored(dir) is true, if the adjacent cell in direction dir is a cell of the environment, which was not visited yet.

Algorithm 1.2 DFS

DFS:

        Choose *dir*, such that reverse(*dir*) is a boundary cell;
        ExploreCell(*dir*);

ExploreCell(*dir*):

        // Left-Hand-Rule:
        ExploreStep(ccw(*dir*));
        ExploreStep(*dir*);
        ExploreStep(cw(*dir*));

ExploreStep(*dir*):

        if unexplored(*dir*) **then**
          move(*dir*);
          ExploreCell(*dir*);
          move(reverse(*dir*));
        end if
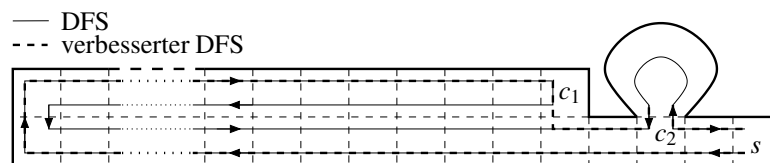
Figure 1.10: First simple improvement of DFS.

A first simple improvement for DFS is as follows:

> If there are no unexplored adjacent cells around the current cell, move back along the shortest path (use all already explored cells) to the *last* cell, that still has an unexplored neighbouring cell.

Figure 1.10 sketches this idea: After visiting c_1 the pure DFS will backtrack along the full corridor of width 2 and reach cell c_2 where still something has to be explored. With our improvement we move directly from c_1 to c_2. Note that for the shortest path we can only make use of the already visited cells. We have no further information about the environment.

By this argument we no longer use the step "move(reverse(*dir*))" in the procedure ExploreStep. After the execution of ExploreCell we can no longer conclude that the agent is on the same cell as before. Therefore we store the current position of the agent and use it as a parameter for any call of ExploreStep. The function unexplored(*base*, *dir*) gives "True", if w.r.t. cell *base* there is an unexplored adjacent cell in direction *dir*. We re-formalize the behaviour as follows:

Algorithm 1.3 DFS with optimal return trips

DFS:

        Choose dir, such that reverse(dir) is a boundary cell;
        ExploreCell(dir);
        Move along the shortest path to the start;

ExploreCell(dir):

        $base$:= current position;
        // Left-Hand-Rule:
        ExploreStep($base$, ccw(dir));
        ExploreStep($base$, dir);
        ExploreStep($base$, cw(dir));

ExploreStep($base$, dir):

        if unexplored($base$, dir) **then**
          Move along the shortest path
            among all visited cells to $base$;
          move(dir);
          ExploreCell(dir);
        end if
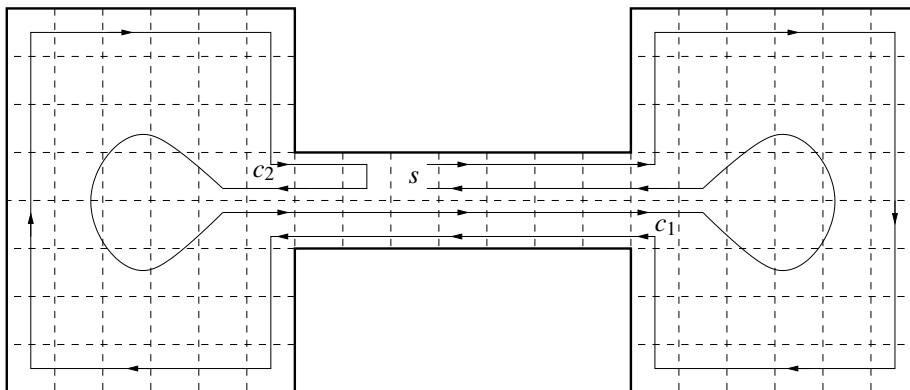
Figure 1.11: Second improvement of DFS.

Algorithm 1.4 SmartDFS

SmartDFS:

        Choose direction *dir*, such that reverse(*dir*) is a boundary cell;
        ExploreCell(*dir*);
        Move along the shortest path to the start;

ExploreCell(*dir*):

        Mark the cell with its layernumber;
        base := current Position;
        if not SplitCell(*base*) **then**
          // Left-Hand-Rule:
          ExploreStep(*base*, ccw(*dir*));
          ExploreStep(*base*, *dir*);
          ExploreStep(*base*, cw(*dir*));
        else
          // Choose different order:
          Calculate the type of the components by the layernumbers
            of the surrounding cells;
          if No component of typ (III) exists **then**
            Move one step by the Right-Hand-Rule;
          else
            Visit the component of type (III) last.
          end if
        end if

ExploreStep(*base*, *dir*):

        if unexplored(*base*, *dir*) **then**
          Move along the shortest path along
            the visited cells to *base*;
          move(*dir*);
          ExploreCell(*dir*);
        end if

For a second kind of improvement we consider the gridpolygon Figure 1.11. In this example the current DFS variant fully surrounds the polygon. Finally the agent has to move back from c_2 to c_1 so that the corridor of width 2 is visited almost 4 times. Obviously it would be better to first fully explore the component at c_1 move to the other component at c_2 and finally move back to the start. In this case the critical corridor will be visited only once. So, if the exploration splits the polygon into components that have to be considered, we have to take care which component should be visited first.

A cell (like the cell c_1) where the remaining polygon definitely splits into different parts is called a **split-cell**. At the first visit of split-cell c_1 in Figure 1.11 it seems to be better to not apply the Left-Hand preference. This depends on the location of the starting point, because we have to move back at the end. The idea can be formulated as follows.

> If the unexplored part of the polygon definitely is splitted into different components (i.e., the graph of unexplored cells is splitted into different components), try to visit the unexplored part that does not contain the starting point.

This idea leads to the Algorithm 1.4 (SmartDFS). It remains to decide, which component actually *contains* the starting point. For this we introduce some notions. Until the first split happens we apply the Left-Hand-Rule and successively explore the polygon layer by layer from the outer boundary to the inner parts. We require a formal definition of the layers.

$2l$ Kanten dazu

π

l

l $2l$ Kanten weniger
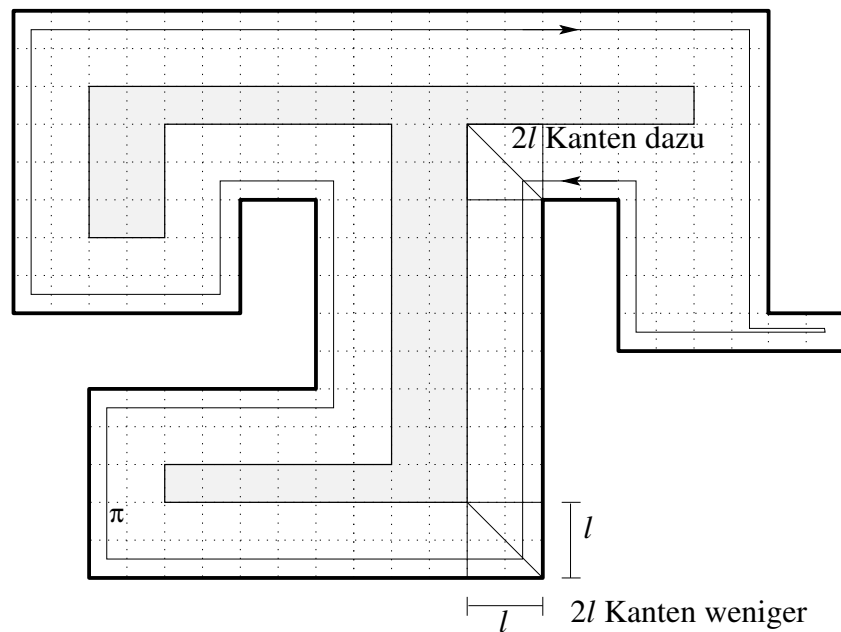
Figure 1.12: The ℓ-Offset of gridpolygon P.

Definition 1.8   Let P be a (simple) gridpolygon. The cells of P that share a boundary edge belong to the first layer, the **1-Layer** of P. The gridpolygon that stems from P without the 1-Layer is called the **1-Offset** of P. Recursively, the **2-Layer** of P, is the **1-Layer** of the **1-Offset** of P and the **2-Offset** of P is the **1-Offset** of the **1-Offset** of P and so on.

Note that the ℓ-Offset of a gridpolygon need not be connected and finally the Offsets will decrease to an empty polygon. The definition is totally independent from any strategy. Fortunately, during the execution of SmartDFS on a simple gridpolygon, we can successively mark and store the layers for any visited cell. The ℓ-Offset has an interesting property.

Lemma 1.9   *The non-empty ℓ-Offset of a simple gridpolygon P has at least 8ℓ edges less than P.*

Proof. We surround the boundary of the gridpolygon in clockwise order and visit all boundary edges along this path. Let us assume that the offset remains a single component. For a left turn the ℓ-Offset 2ℓ looses 2ℓ edges for a right turn the ℓ-Offset 2ℓ wins 2ℓ edges We can show that there are 4 more right turns than left turns. So the ℓ-Offset has at least 8ℓ edges less than P. Even more edges will be cancelled, if the polygon fell into pieces. □

Exercise 5 *Show that for any surrounding of the boundary of a simple gridpolygon in clockwise order there are 4 more rigth turns than left turns. Make use of induction.*

Exercise 6 *Show that in the above proof the non-empty ℓ-Offset will loose even more edges, if it consists of more than one connected component. Show the statement for the 1-Offset.*
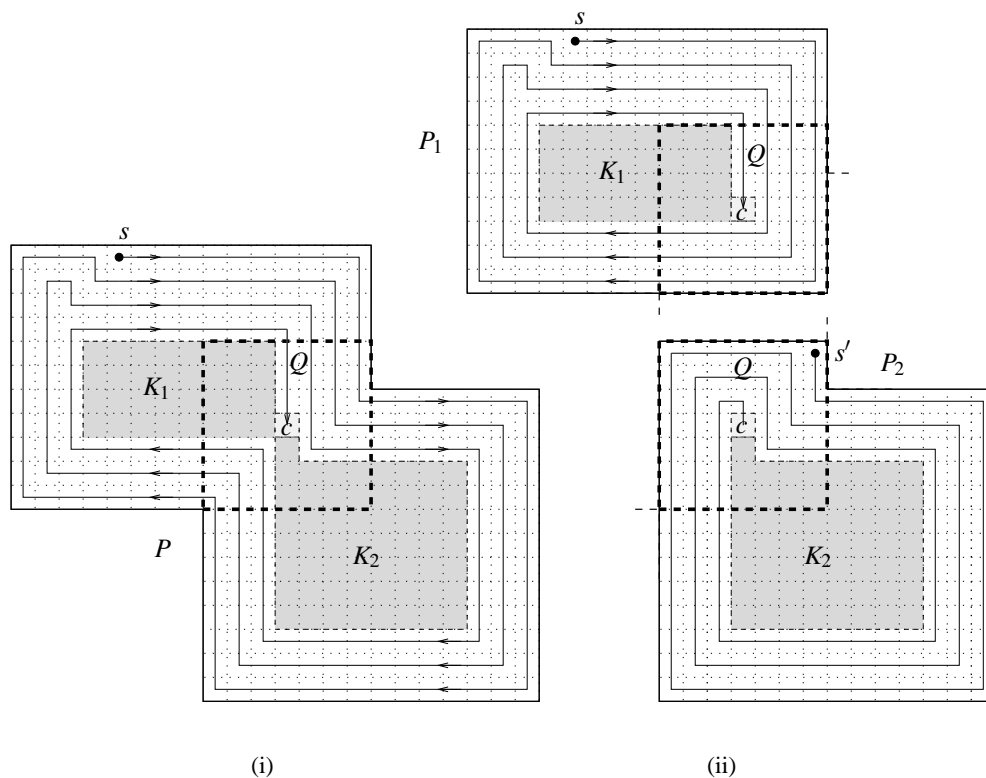
Figure 1.13: Decomposition at a split-cell.
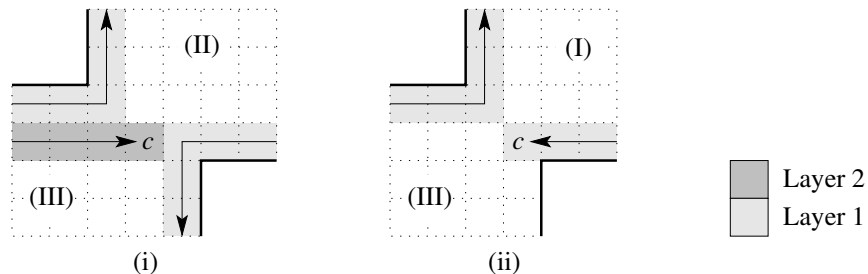
Figure 1.14: Three types of components.

We consider Figure 1.13(i): In the 4. Layer for the first time a split-cell c occurs. Now we decompose the polygon into different components[2]:

---

[2]Let $A \overset{.}{\cup} B$ denote the **disjoint union** $A \overset{.}{\cup} B = A \cup B$ mit $A \cap B = \emptyset$.

$$P = K_1 \,\dot\cup\, K_2 \,\dot\cup\, \{\text{ visisted cells of } P\},$$

where K_1 denotes the component that was visited last. SmartDFS recursively works on K_2, returns to c and proceeds with K_1.

By the layernumbers we would like to avoid the situation of Figure 1.11. We will find the split-cell in layer ℓ, which gives three types of components; see Figure 1.14:

(I)   Component K_i is *fully* surrounded by layer ℓ.

(II) Component K_i is *not* surrounded by layer ℓ (may be touched by the split-cell only).

(III) Component K_i is *partly* surrounded by layer ℓ (not only touched by the split cell).

Obviously, if a split-cell occurs, we should visit the component of type (III) last because the starting point lies in the outer layers of this component.

(i)   (ii)
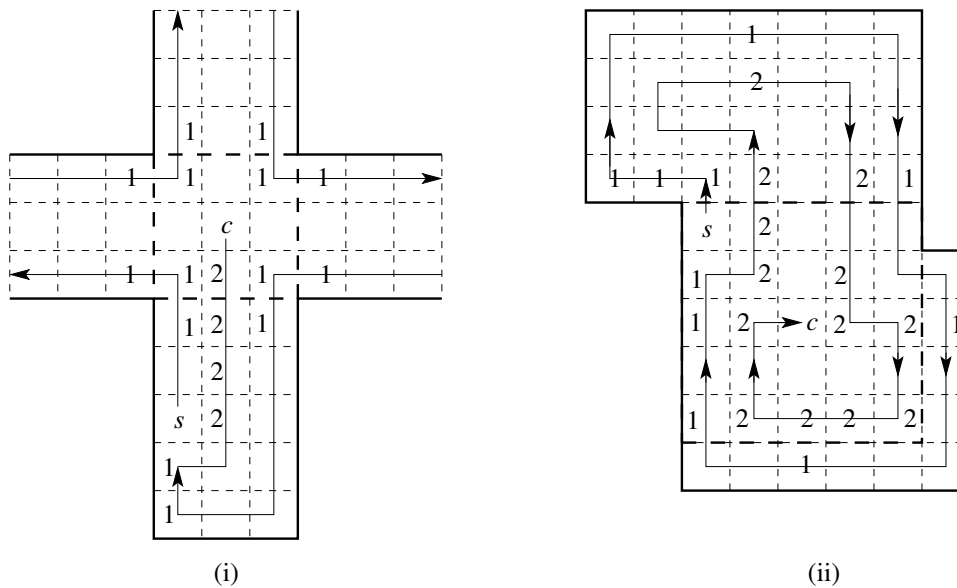
Figure 1.15: Special cases: No component of typ (III) exists.

There are some situations where the a component of type (III) does not exist. For example if the split-cell is the first cell on the next layer, or the component of the starting point was just explored (efficiently). More precisely:

(a) The component with the starting point on its layer was just fully explored in the current layer; see Figure 1.15(i). In this case the order of visiting the remaining components is not critical, we can choose an arbitrary order. This example also shows that at a split-cell more than two components has to be visited. We simply apply one the next step by changing to the Right-Hand-Rule.

(b) Two components have been fully surrounded, because at the split-cell we change from layer ℓ to $\ell + 1$; see Figure 1.15(ii). In all other cases at least one additional visited cell is marked with layer number of the split-cell. We can conclude that layer ℓ was closed with the split-cell. This means that the starting point is not part of the layer of the component where the agent currently comes from. Because the agent normally moves by the Left-Hand-Rule, it suffices to apply the Right-Hand-Rule in this case also.

Altogether in both cases we simply apply the Right-Hand-Rule for a single step.

For the overall analysis at a split-cell we consider two polygons P_1 und P_2 as depicted in Figure 1.13(i). Here we detect the component of type (III). K_2 is a component of type (II). Let Q be a

rectangle of edge length (width or height) $2q+1$ around the split cell c so that

$$q := \begin{cases} \ell, & \text{if } K_2 \text{ has type (I)} \\ \ell-1, & \text{if } K_2 \text{ has type (II)} \end{cases}.$$

Now choose P_2 so that $K_2 \cup \{c\}$ is the q-Offset of $K_2 \cup \{c\}$. The idea is that the rectangle Q will be *added* so that P_2 has the desired form. Now let $P_1 := ((P \backslash P_2) \cup Q) \cap P$, comp. Figure 1.13. The intersection with P is necessary, since there are cases where Q does not totally fit into P. We would like to apply arguments recursively for P_2 and P_1. Let us consider them separately as shown in Figure 1.13(ii). We have choosen P_1, P_2 und Q in a way so that the paths in $P_1 \backslash Q$ und $P_2 \backslash Q$ did not change w.r.t. the paths already performed for P^3. The already performed paths that lead in P from P_1 to P_2 and from P_2 to P_1 will be used and adapted so that the paths outside Q will not change; see Figure 1.13. We can consider P_1 and P_2 separately.

We know that any cell has to be visited at least once. Therefore we count the number of steps $S(P)$ for polygons P as follows. It is the sum of the cells, $C(P)$, of P plus the extra cost $excess(P)$ for the overall *path* length.

$$S(P) := C(P) + excess(P).$$

The following Lemma gives an estimate for the extra cost w.r.t. the above decomposition around a split-cell.

Lemma 1.10 *Let P be a gridpolygon, c a split-cell, so that two remaining components K_1 und K_2 has to be considered. Assume that K_2 is visited first. We conclude:*

$$excess(P) \le excess(P_1) + excess(K_2 \cup \{c\}) + 1.$$

Proof. The agent is located at cell c and decides to explore $K_2 \cup \{c\}$ starting from c and return to c. This gives additional cost at most $excess(K_2 \cup \{c\})$, note that the part $P_2 \backslash (K_2 \cup \{c\})$ can only help for the return path. Because c was already visited, we count one additional item for the excess of visited cells. After that we proceed with the exploration of P_1 and require $excess(P_1)$ for this part. $\qquad\square$

For the full analysis of SmartDFS we have to prove some structural properties:

Lemma 1.11 *The shortest path between to cells s and t in a simple gridpolygon P with $E(P)$ boundary edges consists of at most $\frac{1}{2}E(P) - 2$ cells.*

Proof. W.l.o.g. we assume that s and t are in the first layer, otherwise we can choose different s or t whose shortest path is even a bit longer. Consider the path, π_L, in clockwise order in the first layer from s to t and the path, π_R, in counter-clockwise order in the first layer from s to t. Connecting π_L and π_R gives a full roundtrip. As in the proof of Lemma 1.9 counting the edges gives 4 more edges than cells which gives

$$|\pi_R| + |\pi_L| \le E(P) - 4$$

visited cells.

In the worst case both path have the same length, which gives $|\pi(s,t)| = |\pi_R| = |\pi_L|$, and $2|\pi(s,t)| \le E(P) - 4 \Rightarrow |\pi(s,t)| \le \frac{1}{2}E(P) - 2$. $\qquad\square$

Lemma 1.12 *Let P be a gridpolygon and let c be a split-cell. Define P_1, P_2 und Q as above. For the number of edges we have:*

$$E(P_1) + E(P_2) = E(P) + E(Q).$$

---

[3]For the uniqueness of this decomposition into P_1 and P_2 we remark that P_1 and P_2 are connected, respectively and $P \cup Q = P_1 \cup P_2$ and $P_1 \cap P_2 \subseteq Q$ holds.

Proof. For arbitrary gridpolygons P_1 and P_2 we conclude

$$E(P_1) + E(P_2) = E(P_1 \cup P_2) + E(P_1 \cap P_2).$$

Let $Q' := P_1 \cap P_2$, we have:

$$
\begin{aligned}
E(P_1) + E(P_2) &= E(P_1 \cap P_2) + E(P_1 \cup P_2) \\
&= E(Q') + E(P \cup Q) \\
&= E(Q') + E(P) + E(Q) - E(P \cap Q) \\
&= E(P) + E(Q), \text{ since } Q' = P \cap Q
\end{aligned}
$$

$\square$

Exercise 7 *Show that for arbitrary two gridpolygons P_1 and P_2 we have $E(P_1) + E(P_2) = E(P_1 \cup P_2) + E(P_1 \cap P_2)$.*

Using all these arguments we can show:

Theorem 1.13 *(Icking, Kamphans, Klein, Langetepe, 2000)*
For a simple gridpolygon P with C cells and E boundary edges the strategy SmartDFS required no more than

$$C + \frac{1}{2}E - 3$$

for the exploration of P (with return to the start). This bound will be attained exactly in some environments.
[IKKL00b]

Proof. By the above arguments it suffices to show $excess(P) \leq \frac{1}{2}E - 3$. We give a proof by induction on the number of components.

Induction base:

Assume that there is no split-cell. For the exploration of a single component, SmartDFS visits all cells exactly once and return to the start. For visiting all cells we require $C - 1$ steps. Now the excess is the shortest path back. By Lemma 1.11 $\frac{1}{2}E - 2$ steps suffices which gives the conclusion

Induction step:

Consider the (first) decomposition at a split-cell c. Let K_1, K_2, P_1, P_2, Q be defined as above, assume that K_2 is visited last. We have:

$$
\begin{aligned}
excess(P) &\leq\ excess(P_1) + excess(K_2 \cup \{c\}) + 1 \text{ (Lemma 1.10)} \\
&\leq_{\text{(I.A.)}}\ \frac{1}{2}E(P_1) - 3 + \frac{1}{2}\underbrace{E(K_2 \cup \{c\})}_{\leq E(P_2) - 8q \text{ (Lemma 1.9)}} - 3 + 1 \\
&\leq\ \frac{1}{2}\Big[\underbrace{E(P_1) + E(P_2)}_{\leq E(P) + 4(2q+1) \text{ (1.12, Def. of } Q)}\Big] - 4q - 5 \\
&\leq\ \frac{1}{2}E(P) - 3
\end{aligned}
$$

$\square$

A Java-Applet for the Simulation of SmartDFS and different strategies can be found at:

    http://www.geometrylab.de/

Finally, we would like to show, how to compute the offline shortest paths in gridpolygons Of course the Dijkstra algorithm can also be applied on the gridgraph, but this algorithm does not use the grid structure directly. As an alternative we apply Algorithm 1.5 (C. Y. Lee, 1961, [Lee61]), the running time is only linear in the number of overall cells. The algorithm simulates a wave propagation starting from the goal. Any cell will be marked with a label indicating the distance to the goal. Obstacles *slow down* the propagation a bit; see Figure 1.16. When the wave reaches the starting point s, we are done with the first phase. For computing the path we start at s and move along cells with strictly decreasing labels. Obviously, the shortest path need not be unique.

Algorithm 1.5 Algorithm of Lee

Shortest path from s to t in a gridpolygon

   Datastructur: Queue Q
   // Initialise
   Q.InsertItem(t);
   Mark t with label 0;
   // Wave propagation:
   loop
      $c := Q$.RemoveItem();
      for all Cells x such that x is adjacent to c and x is not marked **do**
         Mark x with the label of $label(c) + 1$;
         Q.InsertItem(x);
         if $x = s$ **then** break loop;
      end for
   end loop
   // Backtrace:
   Move along cells with strongly decreasing labels from s to t.
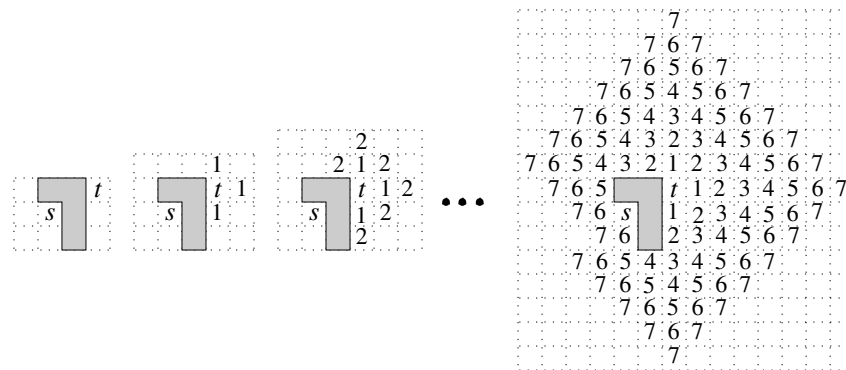
Figure 1.16: Wave-Propagation.

1.4.2   Competitive ratio of SmartDFS

The corridor of width 3, see Figure 1.7, indicates that the competitive ratio of SmartDFS should be better than 2. SmartDFS runs 4 times though the corridor whereas the shortest path visits any cell only once. This gives roughly a ratio of $\frac{4}{3}$. We will show that this is the worst-case for SmartDFS. The gap between e $\frac{7}{6}$ and $\frac{4}{3}$ is small.

For the analysis we first give a precise definition of the structure of parts of gridpolygons which will be explored in an optimal fashion. The SmartDFS Strategy does not make any detours within these passages.

For a *corridor* of widths 1 this is abviously true. But also corridors of width 2 will be passed optimally, since SmartDFS runs forth and back along different tracks; see Figure 1.17. We give a formal definition of the *narrow passages*.

Definition 1.14   The set of cells that can be deleted such that the layernumber of the remaining cells do not change are called narrow passages of P.
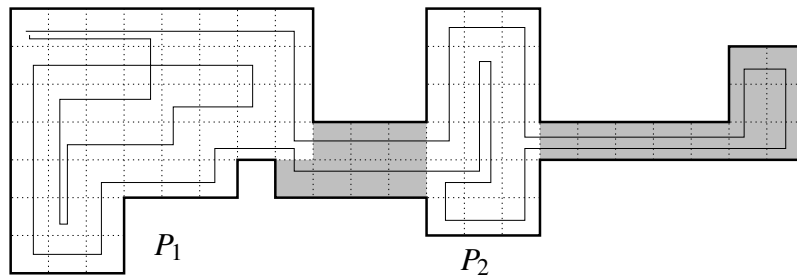
Figure 1.17: SmartDFS is optimal in narrow passages.

SmartDFS passes narrow passages optimally since they allow an optimal forth and back pass-through. There are no additional detours at the entrance and exit of a narrow passage because they consist of cells in the first layer. They can be considered as *gates*. The entrance and exit is always precisely determined.

The idea is to consider polygons without narrow passages first. There is a fixed relationship between edges and cells.

Lemma 1.15   *Let P be a simple gridpolygon without narrow passages and without a split-cell in the first layer. We have*

$$E(P) \leq \frac{2}{3}C(P) + 6.$$

Proof. A 3×3 gridpolygon has precisely this property, $C(P) = 9$ und $E(P) = 12$. Any gridpolygon with the above conditions can be reduced by successively removing columns or rows such that in each step the property remains true and such that always at least 3 cells and at most 2 edges will be removed. This is an exercise below.

Starting backwards from the property $E(P_0 = \frac{2}{3}C(P_0) + 6$ we will maintain the bound $E(P_i) \leq \frac{2}{3}C(P_i) + 6$ since we add at least 3 cells and add at most 2 edges. Finally, $E(P) \leq \frac{2}{3}C(P) + 6$ holds.

$\square$

First, we show that the overall number of exploration steps of SmartDFS decreases for the given class of polygons.

Lemma 1.16   *A simple gridpolygon P with $E(P)$ edges and $C(P)$ cells, without narrow passages and without a split-cell in the first layer well be explored by SmartDFS with no more than $S(P) \leq C(P) + \frac{1}{2}E(P) - 5$ steps.*
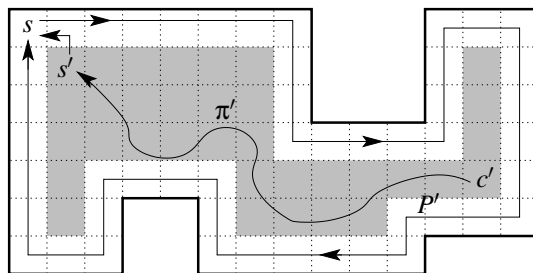
Figure 1.18: A simple gridpolygon without narrow passages and no split-cell in the first layer has the property $E(P) \leq \frac{2}{3}C(P) + 6$. After the first coil SmartDFS starts in the 1-Offset P'. The return path to c' from an arbitrary point in P' is shorter than $\frac{1}{2}E(P)/2 - 2$.

Proof. From Theorem 1.13 we conclud $S(P) \leq C(P) + \frac{1}{2}E(P) - 3$. By the properties of P, SmartDFS performs a full first round from s to the first cell s' in the second layer. After that, in principle we start SmartDFS again at s' in a gridpolygon (1-Offset P' of P); see P' in Figure 1.18. P' is path connected and by Lemma 1.9 P' has 8 edges less than P;

The cells in the first layer have been visited optimally the path length from s to s' coincidence with the number of cells in the first layer. Finally, we have to count two additional steps from s' to s. Altogether, we require $S(P) \leq C(P) + \frac{1}{2}(E(P) - 8) - 3 + 2 = C(P) + \frac{1}{2}E(P) - 5$ steps. □

With the statements above we will be able to prove the main result.

Mit diesen Vorbereitungen können wir die kompetitive Schranke beweisen.

Theorem 1.17 *(Icking, Kamphans, Klein, Langetepe, 2005) The SmartDFS strategie for the exploration of simple gridpolygons is $\frac{4}{3}$-competitive!* [IKKL05]

Proof.

Let P be a simple gridpolygon. First, we remove the narrow passages from P. We know that the entrance and exits over the gates by SmartDFS are optimal. We obtain a sequence P_i, $i = 1, \dots, k$ of gridpolygons connected by narrow passages. See for example P_1 and P_2 in Figure 1.17.

We can consider the gridpolygons P_i separately. We can also assume different starting points. The movement between the gates count for the required additional steps. It is sufficient to show $S(P_i) \leq \frac{4}{3}C(P_i) - 2$ for any subpolygon. This bound exactly holds for $3 \times m$ gridpolygons for even m; see Figure 1.19.

We show the bound by induction over the number of splil-cells.

SmartDFS   optimal strategy
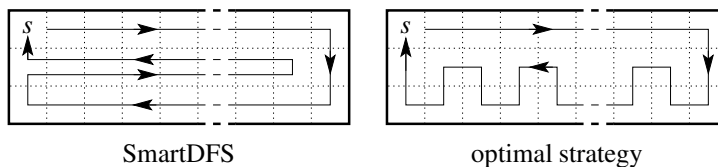
Figure 1.19: In a corridor of width 3 and with even lenght the bound $S(P) = \frac{4}{3}S_{\text{Opt}}(P) - 2$ holds.

Induktion-Base: If P_i has no split-cell, there is also no split-cell in the first layer. We apply Lemma 1.16 and Lemma 1.15 and obtain:

$$S(P_i) \leq C(P_i) + \frac{1}{2}E(P_i) - 5$$

$$\leq \quad C(P_i) + \frac{1}{2}\left(\frac{2}{3}C(P_i) + 6\right) - 5$$

$$= \quad \frac{4}{3}C(P_i) - 2.$$

Induktion-Step: If there is no split-cell in the first layer we can apply the same arguments as above. Therefore, we assume that the first split occurs in the first layer. Two cases can occur as depicted in Figure 1.20.

In the first case the component of type (II) was not visited before and we define $Q := \{c\}$. The second case occurs, if the split-cell c is diagonally adjacent to a cell c'; compare Figure 1.20(ii), (iii) and (iv). We build the smallest rectangle Q that contains c and c'. In case (ii) and (iii) Q is a square of size 4. In case (iv) by simple adjacency Q is a rectangle and $|Q| = 2$.

Analogously to the proof of Theorem 1.13 we split the polygons into parts P' and P'' both containing Q.

Here P'' is of type (I) or (II) and P' the remaining polygon. das Polygon der Komponenten vom Typ (I) oder (II) und P' das andere.

For $|Q| = 1$ (see Figure 1.20(i)) we have $S(P_i) = S(P') + S(P'')$ and $C(P_i) = C(P') + C(P'') - 1$. We apply the induction hypothesis on P' and P'' (they have one split-cell less) and obtain:

$$\begin{aligned}
S(P_i) \quad &= \quad S(P') + S(P'') \\
&\leq \quad \frac{4}{3}C(P') - 2 + \frac{4}{3}C(P'') - 2 \\
&\leq \quad \frac{4}{3}C(P_i) + \frac{4}{3} - 4 \; < \; \frac{4}{3}C(P_i) - 2.
\end{aligned}$$

For $|Q| = 4$ we argue that by the union we will save some steps that will occur for the separate explorations. We consider P' and P'' separately, first. The movements from c' to c (and c to c') count in both polygons. For the complete P_i the path from c' to c (and c to c') are given either P' or in P'', this means that we save $4 = |Q|$ steps.

We have $S(P_i) = S(P') + S(P'') - 4$ and $C(P_i) = C(P') + C(P'') - 4$. By induction hypothesis for P' and P'' we conclude:

$$\begin{aligned}
S(P_i) \quad &= \quad S(P') + S(P'') - 4 \\
&\leq \quad \frac{4}{3}C(P') + \frac{4}{3}C(P'') - 8 \\
&= \quad \frac{4}{3}\left(C(P') + C(P'') - 4\right) - \frac{8}{3} \\
&< \quad \frac{4}{3}C(P_i) - 2.
\end{aligned}$$

The case $|Q| = 2$ is left as an exercise.

Altogether an optimal strategy requires $\geq C(P_i)$ steps or $\geq C(P)$ in total and we have a competitive ratio of $\frac{4}{3}$. $\square$

Exercise 8 *Analyse the remaining case $|Q| = 2$ in the above proof.*

If we compare the result to Theorem 1.7 there is a gap of size $\frac{1}{6}$ between $\frac{7}{6}$ and $\frac{4}{3}$. Recently, both parts have been improved. There is a lower bound of $\frac{20}{17}$ and an upper bound of $\frac{5}{4}$ shown by Kolenderska et. al 2010. In principle the strategy is a local improvement of SmartDFS and the lower bound is an extension of our construction. The result comes along with a tedious case analysis.
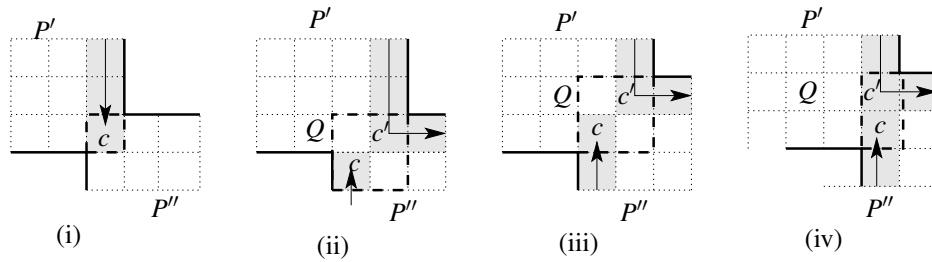
Figure 1.20: A gridpolygon P_i that is separated into components of type (I) or (II) at the split-cell. The rectangle Q is always inside P_i.
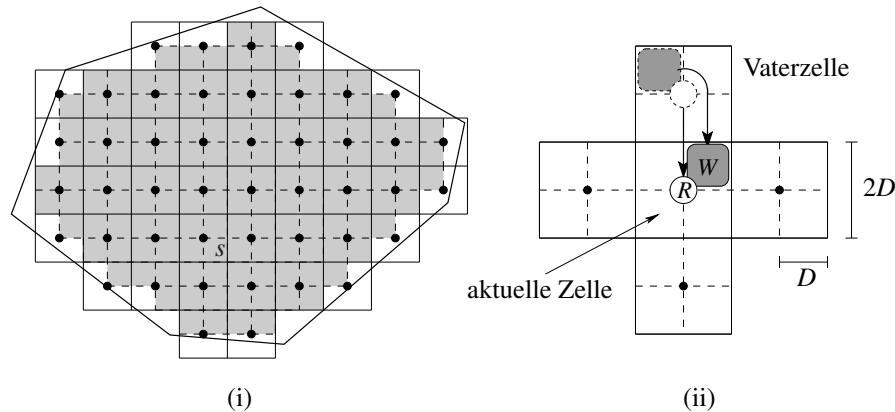
Figure 1.21: $2D$-cells and $D \times D$ sub-cells.

1.4.3   Exploration of general gridpolygons

For the more general exploration of gridpolygons we first slightly change the model[4]: We consider an agent that is located at the center of 4 cells of size $D \times D$. The tool for the exploration still has size $D \times D$ as before and moves freely around the agent. More precisely, we consider 4 **sub-cells** of size $D \times D$ and unify them to a $2D$-cell[5]; see Figure 1.21(i).

It can happen that for the $2D$-cell, not all sub-cells belong to the initial gridpolygon, since some of the sub-cells simply belong to the boundary. Such $2D$-cell are denoted as **partially occupied cells**.

In Figure 1.21(i) all cells intersected by the original polygonal segments are partially occupied (compare also reffigfigOnline/PolyToGrid on page 8). The agent is always located in the center of the $2D$-cell. Analogously to the SmartDFS model, the agents scans the four adjacent $2D$-cells. The tool moves freely around the agent, we would like the count the number of steps of the tool; see also Figure 1.21(ii).

The current cell of the agent is denoted as *current cell*. The *parent cell* of the agent is the cell where he is actually coming from. In the beginning we initially an arbitrary adjacent $2D$-cell as the parent cell.

The strategy "Spanning-Tree-Covering" (STC) constructs a spanning tree for all connected $2D$-cells that are also not occupied. The tool moves along the spanning tree by the Left-Hand-Rule. The construction can be done fully online. The $2D$-cells are detected by the Right-Hand-Rule. Obviously by this approach the tool exactly visits any cell at most once by following the spanning tree. Figure 1.22(i) shows an example for the efficient exploration of all non-occupied cells by $2D$-Spiral-STC. As mentioned before, for the start we can choose an arbitrary parent cell.

The disadvantage of $2D$-Spiral-STC is, that we do not visit sub-cells by that tool which actually lie in the connected component of the sub-cells. Now we relax the behaviour of $2D$-Spiral-STC. The strategy

---

[4]We will see later that the change was only done for the reason of a convenient analysis and description.
[5]In the following a cell always denotes a $2D$-cell.

Algorithm 1.6 2D-Spiral-STC

2DSPSTC(*parent*, *current* **):**

Mark *current* as visited.

while *current* has unvisited neighbour cell **do**

- From *parent* search in ccw order for a neighbouring cell *free*, which is not marked as visited and is not partially occupied.
- Build the spanning tree edge from *current* to *free*.
- Move the tool by Left-Hand-Rule along the spanning tree edge to the first sub-cell of *free*.
- Call 2DSPSTC(*current*, *free*).

end while

if *current* $\neq s$ **then**

- Move by the Left-Hand-Rule along the spanning tree edge back from *current* to the first sub-cell of *parent*.

end if

Algorithm 1.7 SpiralSTC

SPSTC(*parent*, *current* **):**

Mark *current* as visited.

while *current* has unvisited neighbour cell **do**

- From *parent* search in ccw order for the first neighbouring cell *free*.
- Build a spanning tree edgs from *current* to *free*.
- Move the tool along the spanning tree edge to the first sub-cell of *free*. The movement depends on the local situation. For double-sided edges the tool moves by Left-Hand-Rule along the edge. For single-sided edges the tool might change to the other (left) side of the spanning tree edge in order to avoid an occupied sub-cell for reaching the corresponding sub-cell.
- Call SPSTC(*current*, *free*).

end while

if *current* $\neq s$ **then**

- Move along the spanning tree edge back from *current* to the first possible sub-cell of *parent*. The movement depends on the type of the edge, as mentioned above.

end if

(i)   (ii)
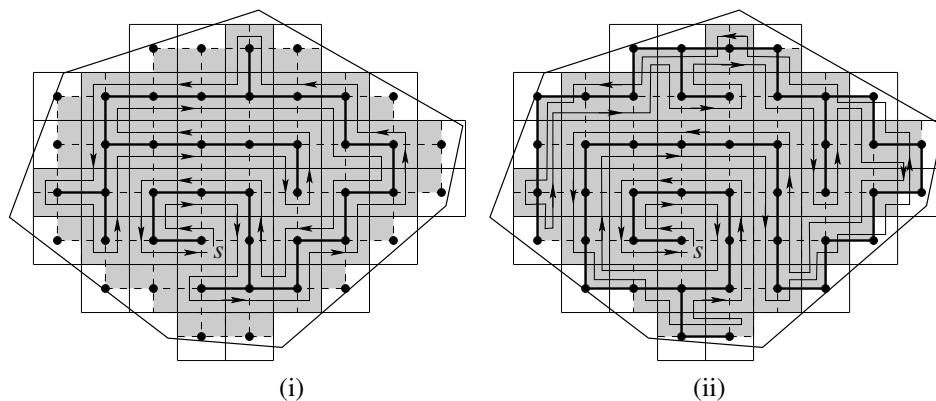
Figure 1.22: Examples for (i) 2*D*-Spiral-STC and (ii) Spiral-STC.

Spiral-STC (Algorithm 1.7) also constructs a spanning tree in an online fashion. But we also insert a corresponding edge if a partially occupied 2*D*-cell contains sub-cells that are still reachable by the tool. In this case the tool cannot always move the the Left-Hand-Rule along the spanning tree edge. The tool has to avoid occupied sub-cells and visits some sub-cells more than once. For systematically analysing the corresponding additional sub-cell visits of the tool we make use of the following notion:

Doppelseitige Kante   Einseitige Kante   Lokal unzusammenh"angende
                                                                Zelle
(i)   (ii)   (iii)
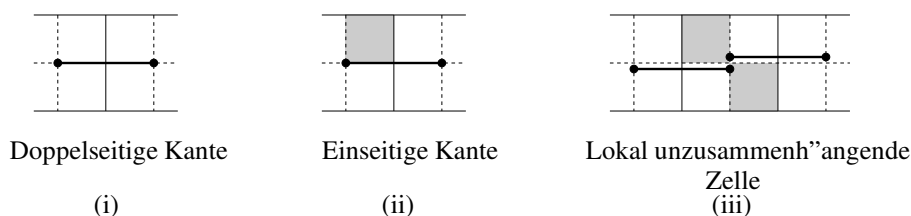
Figure 1.23: (i) Double-sided edge, (ii) one-sided edge, (iii) locally disconnected 2*D*-cell.

Definition 1.18   A spanning tree edge constructed by STC in a gridpolygon *P* is denoted as
  (i) **double-sided edge**, if all adjacent sub-cells belong to the gridpolygon *P* (Figure 1.23(i)),
 (ii) **single-sided edge**, if at least one of the adjacent sub-cells is a boundary sub-cell of *P* (Figure 1.23(ii)).

Double-sided edges are handled in the same way as in the 2*D*-Spiral-STC strategy. Single-sided edges impose a detour for the tool, some sub-cells will be visited more than once since the tool changes to the other side of the spanning tree edge. For the analysis we will consider the corresponding cases systematically. A special case occurs, if the situation imposes two spanning tree edges for the same cell from different directions. The cell is locally disconnected in this case; see Figure 1.23(iii). This 2*D*-cell will be visited twice from different directions. For simplicity we internally double the corresponding vertex and the spanning tree has exactly one incoming edge for any vertex. For the analysis we have to take care that we count the cell only once. An example of the execution of Spiral-STC is shown in Figure 1.22(ii).

By the preference rule for the 2*D*-cells the Spiral-STC constructs spanning trees with many windings. This is not always intended, especially for lawn-mowing or vacuum-cleaning a tool should try to avoid so many turns. The number of turns might also be part of the cost model. The Scan-STC variant has a fixed given preference for vertical or horizontal edges. We would like to make local decision for the construction of spanning tree edges. In our examples we prefer a vertical scan of the gridpolygon. For this we extend the sensor model and allow to have information about all diagonally adjacent 2*D*-cells of a current cell.

The idea is that the construction of a horizontal edge will be postponed, if it is clear that we can also reach the 2*D*-cell by another vertical spanning tree edge. To keep the rule simple we only look ahead as

(i)   (ii)   (iii)
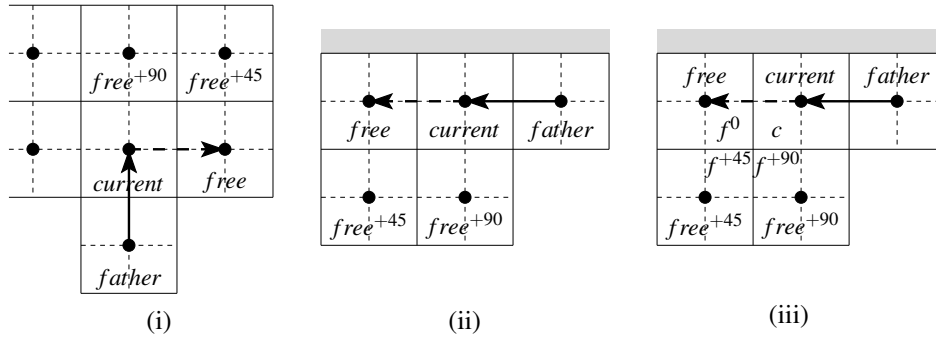
Figure 1.24: Avoid horizontal edges with the Scan-STC.

indicated in Figure 1.24 (i) and (ii). Here we currently would like to build a horizontal edge. The agent is located at cell *current* and is looking (in ccw order starting from *father*) for the first free cell *free*. If there is a counterclockwise path from *free* over $free^{+45}$ and $free^{+90}$ back to the current cell, we change the preference and build a spanning tree edge to $free^{+90}$. Here $free^{+45}$ lies on the sam row as *free* and is the the next cell in ccw order from *free*. $free^{+90}$ is the next cell in ccw order from $free^{+45}$ in the same column as *current*.

If the full turn exists, the cell *free* will also be reached from $free^{+45}$ be a vertical edges and $free^{+45}$ can be reached from $free^{+90}$. Note that we have extended the sensor model in this case and also have information about diagonally adjacent edges.

Analogously, we can also consider partially occupied 2*D*-cells and apply the same idea. For the corresponding avoidance rule we consider the sub-cells c, f^0, f^{+45} and f^{+90} instead if the cells *current*, *free*, $free^{+45}$ und $free^{+90}$; see Figure 1.24(iii).

By the above idea we could define a strategy 2*D*-Scan-STC that corresponds to 2*D*-Spiral-STC. We skip this step and directly define a Scan-STC Algorithm that makes use of the sub-cells c, f^0, f^{+45} and f^{+90} by the same arguments. If f^{+45} and f^{+90} are also free, we will reach f^0 from f^{+45} and in turn f^{+45} from f^{+90}. Algorithm 1.8 summarizes this behaviour.

(i)   (ii)
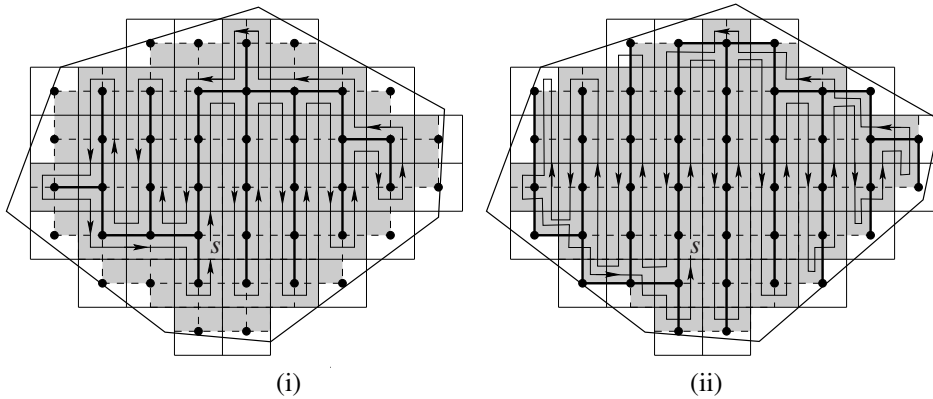
Figure 1.25: Examplle for (i) 2*D*-Scan-STC, (ii) Scan-STC.

Algorithm 1.8 ScanSTC

SCSTC(*parent***,** *current* **):**

  Mark *current* as visited.

  while *current* has unvisited neighbouring cell **do**

      • From *parent* search in ccw order for the fisrt non-visited neighbouring cell *free*.

     if Spanning tree edge from *current* to *free* is horizontal and sub-cells f^{+45} and f^{+90} are free **then**

        $free := free^{+90}$.

     end if

      • Build a spanning tree edge from *current* to *free*.

      • Move the tool along the spanning tree edge to the first sub-cell of *free*. The movement depends on the local situation. For double-sided edges the tool moves by Left-Hand-Rule along the edge. For single-sided edges the tool might change to the other (left) side of the spanning tree edge in order to avoid an occupied sub-cell and reach the corresponding sub-cell.

      • Call SCSTC(*current*, *free*) auf.

  end while

  if *current* $\neq s$ **then**

      • Move along the spanning tree edge from *current* back to the first possible sub-cell of *parent*. The movement depends on the type of the edge, as mentioned above.

  end if

Theorem 1.19 *(Gabriely, Rimon, 2000)*

Let P be a gridpolygon with C sub-cells. Let K be the number of all sub-cells, which are diagonally adjacent to an occupied (boundary) sub-cell[6]. The gridpolygons P will be explored by Spiral-STC and Scan-STC in time $O(C)$ and space $O(C)$. There are no more than

$$S \leq C + K$$

exploration steps, S, for the tool.   *[GR03]*

Proof.

Correctness:

Both algorithms construct a spanning tree by DFS such that any 2*D*-cell which has reachable *D* sub-cells will be visited. The tool moves along the spanning tree on both sides – as long as the path is not blocked – and visits all sub-cells that are *touched* by the spanning tree.

Zelle	"Ubergr.	Intern	Gesamt	Randzellen
1	0	1	1	2
2	1	2	3	3
3	1	2	3	3
4	1	1	2	2
5	1	2	3	3
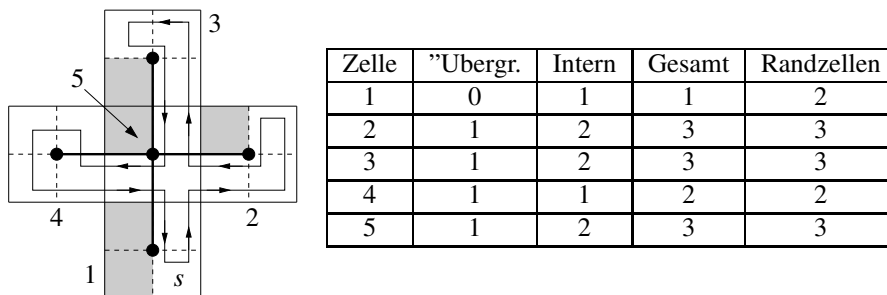
Figure 1.26: Estimating the double visits of sub-cells by STC locally.

Path length:

The number of steps for the tool is essential the sum of the visited sub-cells *C*. If the tool changes to the left side of a spanning tree a detour has to be made and some sub-cells will be visited more than once. Beyond *C* we simply count the number of sub-cells that are visited more than once and locallly charge the sub-cells of a 2*D*-cell for these visits.

---

[6]*K* can be estimated by the number of sub-cells in the first layer of *P*.
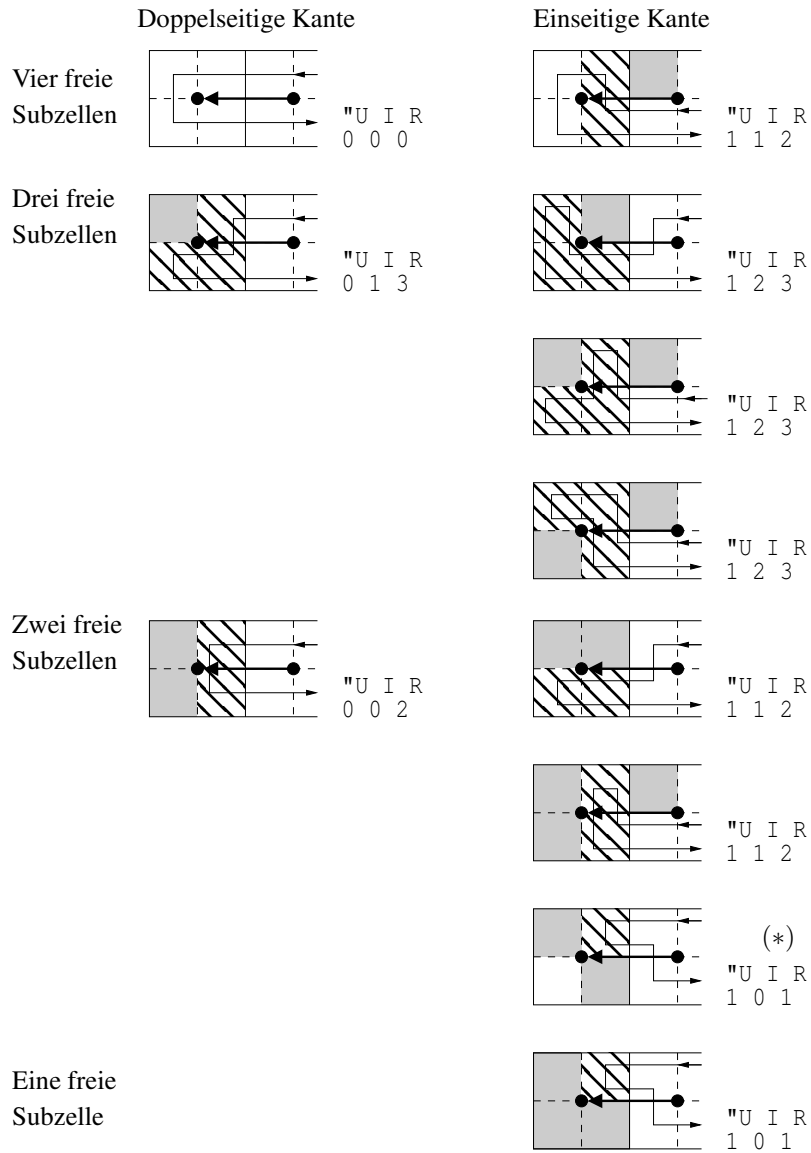
Figure 1.27: Analysis of STC, all possible cases.

We differentiate between *inner* double visits and *intra* double visits. The latter one occur during the movement inside a 2*D*-cell if a sub-cell is visited again. The former one occur if we leave a 2*D* cell *c* along the spanning tree to a neighbouring cell and the corresponding sub-cell was visited before. For this double visit we also charge the 2*D* cell *c*, since it was responsible for the detour.

Any 2*D*-cell *c* is visited for the first time by an incoming spanning tree edge. The inner-cell double visit will occur only if the cell *c* is left again along this edge. Figure 1.26 shows an example for counting inner and intro double visits. For cell 1 sub-cell *s* is visited twice, an intra double visit. The sub-cell above *s* is also visited twice, but by the movement back for 5 to 1 along the spanning tree edge. Therefore 2*D*-cell 5 is charged for this by an inner double visit.

The table of 2*D*-cells Figure 1.26 shows the number of inner and intra double visits for any 2*D*-cell. We charge the 2*D*-cells for these double visits. On the other hand, for any 2*D*-cell we also count the number of sub-cells that are diagonally adjacent to a boundary sub-cell. The corresponding boundary sub-cell need not lie inside the 2*D*-cell itself. Surprisingly, the sum of inner and intra double visits does never exceed the number sub-cells with diagonally adjacent neighbours. This is also given in the table of Figure 1.26.

For a full systematic proof we refer to Figure 1.27. Any 2*D*-cell *c* is visited by some spanning tree

edge for the first time and the inner double visits can only occur on this edge. Therefore it is sufficient to consider the $2D$-cell without other outgoing spanning tree edges. For any intra detours only sub-cells of the current cell are responsible. For the inner detour only the parent cell was responsible.

We distinguish between double sided and single sided edges and between the number of boundary sub-cells inside the corresponding $2D$-cell c. We always count inner and intra double visits and compare the sum to the number of sub-cells adjacent to boundary sub-cells.

For all reasonable cases the sum of double visits is always covered locally by the number sub-cells adjacent to boundary sub-cells. The case marked with $(*)$ is a bit tricky. The corresponding $2D$ cell might also be visited by another spanning tree edge. This is not critical because there is only 1 double visit in this case for each sub-case. They can be handled separately.

Running time and space requirement
The tool performs at most $C + K \leq 2C$ steps. Any movement is computed locally in $O(1)$ time. The corresponding overall information required does not exceed $O(C)$. $\qquad\square$

Finally, we consider the Scan-variants of the STC-Algorithms. We would like to give a rough estimate for the efficiency in avoiding horizontal edges by $2D$-Scan-STC.
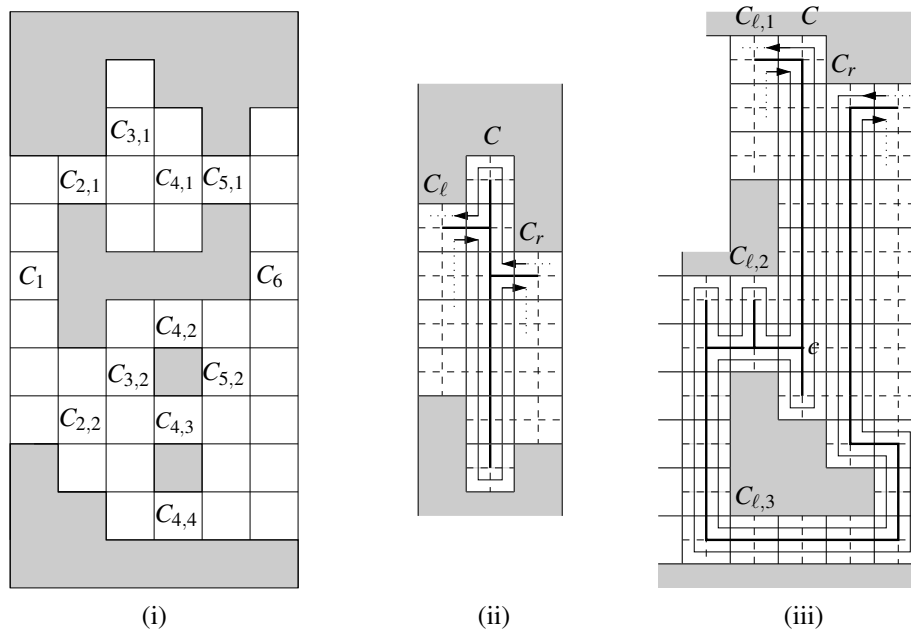
Figure 1.28: (i) Columns and the change of connectivity, (ii) Columns without changes, (iii) Difficult online situation.

We consider columns of the gridpolygon and from left to right we count the change of the connectivity from a column to its neighbour on the right. For example on Figure 1.28(i) there is a numbering of the columns and the number of different vertical components of the columns. From left to right we sum up all differences in the number of components of a column to its neighbour. In Figure 1.28(i) column C_1 has one component and in column C_2 this component split into two components $C_{2,1}$ and $C_{2,2}$. This gives a difference of 1. The components $C_{5,1}$ and $C_{5,2}$ of column C_5 run together in C_6 to a single component. This also is a change of 1 in the difference. Of course also many parts might be involved. We count the changes of any component separately. Let Z denote the sum of all these local changes.

The number Z is a measure for the additional horizontal edges of the spanning tree of Scan-STC against an optimal number of spanning tree edges:

Theorem 1.20 *(Gabriely, Rimon, 2000)*
Let P be a gridpolygon. Let H_{Opt} denote the minimal number of spanning tree edges among all 2D spanning trees of P. Let Z be the above number of connectivity changes for the columns of the 2D-cells.

2D-Scan-STC constructs a spanning tree with at most

$$H_{\text{STC}} \leq H_{\text{Opt}} + Z + 1$$

horizontal edges.   [GR03]

Proof.(Sketch)
If there is no change in a $2D$ column, the optimal spanning tree and $2D$-Scan-STC will visit and leave the column only once; compare Figure 1.28(ii). The main problem is that by $2D$-Scan-STC a connected component of a column will be left by the spanning tree to the same side more than once. This can only happen, if there are changes in the connectivity; see Figure 1.28(iii).   □

Concluding remarks

Arkin, Fekete and Mitchell gave some approximation results for the offline exploration of gridpolygons; see [AFM00]. Betke, Rivest und Singh considered a variant of the exploration problem. They introduced the following piecemeal-condition: The agent has to explore an environment with rectangular obstacles and has to return to the start from time to time (charging an accumulator); see [BRS94]. A strategy for this problem for general grid-environments stems from Albers, Kursawe und Schuierer [AKS02].

1.5 Constrained graph-exploration

We consider the problem of the exploration of an unknown graph $G = (V, E)$ starting from some fixed vertex $s \in V$. This means that we would like to visit all edges and vertices of G. First, we consider unit-weights which means that any visit of an edge has cost 1. Different from the previous section we consider a constrained version of the exploration, due to the following practical models. Let r denote the radius or depth of the graph w.r.t. s. I.e., r is the maximal length of a shortest path from s to some vertex $v \in V$. Let us first assume that r is known, but not the graph itself.

1. The agent is bounded by a tether of length $\ell = (1 + \alpha)r$ (for example a cable constraint).

2. The agent has to return to the start after any $2(1 + \alpha)r$ steps (for example an accumulator has to be recharged).

3. A large graph should be explored up to a given fixed depth d (for example for searching a close by target).

The above third variant will be applied to a searching heuristic with increasing depth, later. First, we show some simple simulation resutls. If an algorithm for the tether variant is known, one can also establish an accumulator strategy with some extra cost.

Lemma 1.21 *Given a tether variant strategy with tether length $l = (1 + \alpha)r$ and overall cost T. For any $\beta > \alpha$ there is an accumulator-strategy with cost $\frac{1+\beta}{\beta-\alpha}T$*

Proof. We design the accumulator strategy by following the tether strategy. After any $2(\beta - \alpha)r$ steps we move back from the current vertex v to the start, recharge the agent and move back to v. Then we proceed with the next step of length $2(\beta - \alpha)r$ of the tether strategy path. In the tether strategy for any vertex v, we are never more than $(1 + \alpha)r$ away from the start. That is $2(\beta - \alpha)r + 2(1 + \alpha)r = 2(1 + \beta)r$ always result in correct loops. The strategy is correct.

On the other hand, we have cost T for following the tether path and additional cost for moving back and force. We move back at most $\frac{T}{2(\beta-\alpha)r}$ times and require $2(1 + \alpha)r$ steps for any movement. This gives total cost:

$$T + \frac{T}{2(\beta - \alpha)r} \cdot 2(1 + \alpha)r = T\frac{\beta - \alpha + 1 + \alpha}{\beta - \alpha} = \frac{1 + \beta}{\beta - \alpha}T.$$

$\square$

Exercise 9 *Given an accumulator strategy S with accumulator size $2(1 + \beta)r$ and overall cost T. For a given $\alpha > \beta$ design an efficient tether strategy that makes use of S so that the cost of the tether strategy is $f(\alpha, \beta) \cdot T$. Determine $f(\alpha, \beta)$ precisely.*

We can also consider the Offline-variant of the problem. In this case the graph is fully known. To the best of our knowledge the complexity of the Offline-variant (computing the best strategy) is not known. Since in the case that the tether is very long, the Hamiltonian-path problem appears to be part of the problem, the problem is assumed to be NP-hard.

If the optimal Offline-strategy is not known, we can design an Offline-strategy that approximates the optimal strategy. We consider the accumulation variant and assume that the accumulator has size $4r$.

Lemma 1.22 *Let us assume that an accumulator of size $4r$ is given. There is a simple Offline algorithm that explores a graph of depth r with no more than $6|E|$ steps.*

Proof. We consider the DFS walk among the edges of the graph which requires $2|E|$ steps. Now we split this overall path into pieces of size $2r$. Similarly to the simulation in the proof above we successively move to the start vertices of these subpaths, follow the DFS path for $2r$ steps and return to the start after that. In total the accumulator of size $4r$ is sufficient. Moving along the DFS path gives $2|E|$ steps. There are no more than $\left\lceil \frac{2|E|}{2r} \right\rceil$ sub-paths that require no more that $\left\lceil \frac{|E|}{r} \right\rceil 2r$ steps in total. We have $\left\lceil \frac{|E|}{r} \right\rceil 2r \le \left(\frac{|E|}{r} + 1 \right) 2r \le 2|E| + 2r$ which shows that $4|E| + 2r \le 6|E|$ is sufficient. $\square$

From now on we consider only the tether variant, for the accumulation variant similar results can be shown. A first simple idea is to take the tether length for the DFS walk into account.

Just performing DFS is not always possible. A BFS approach is always possible but results in too many exploration steps; see Figure 1.29. Therefore we apply DFS with the tether restriction as given in Algorithm 1.9. There is a backtracking step, if the tether is fully exhausted. We call this algorithm bDFS for bounded DFS.
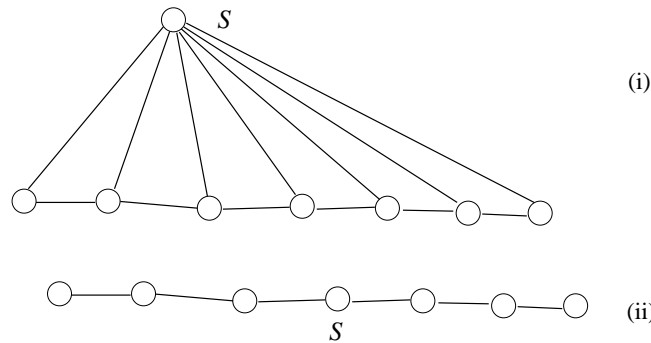
Figure 1.29: (i) A Graph with n vertices and with depth $r = 1$, pure DFS would require a tether of length $n-1$. (ii) A graph of depth n, BFS with a tether of length n requires $\Omega(n^2)$ steps.
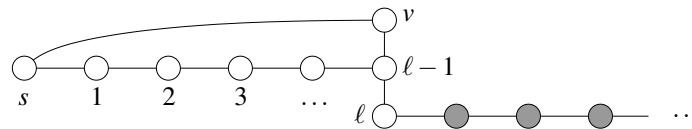
Figure 1.30: bDFS kann einige Knoten nicht erreichen.

Algorithm 1.9 boundedDFS

---

bDFS(v, ℓ):

  if $(\ell = 0) \lor$ (no adjacent non-explored edges) **then**
    RETURN
  end if
  for all non-explored edge $(v, w) \in E$ **do**
    Move from v to w along (v, w).
    Mark (v, w) as *explored*.
    bDFS($w, \ell - 1$).
    Move back from w to v along (v, w).
  end for

In general bDFS is not sufficient for the full exploration of a graph. For example in Figure 1.30 we have the problem that the dark-colored vertices cannot be reached, if the algorithm first chooses the path along the vertices $1, 2, \ldots, \ell - 1$, visits vertex l, v and s and winds back to the start s. The path from s over v is short enough but will not be considered by bDFS.

Therefore we would like to call bDFS from different sources. The aim is to achieve a constant competitive algorithm. In Algorithm 1.10 we maintain a set of (edge) disjoint trees $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ with root vertices $s_1, s_2, \ldots, s_k$, respectively. The trees still contain incomplete vertices where not all adjacent edges have been visited. We choose a tree T_i with the minimal distance from s to root s_i among all trees of $\mathcal{T}$. From this tree we prune subtrees T_{w_j} with root vertices w_j, so that w_j is a certain distance ($minDist = \frac{\alpha r}{4}$) away from s and T_{w_j} has a certain minimal depth (determined over $minDepth = \frac{\alpha r}{2}$). Those trees will be inserted into $\mathcal{T}$. The pruning forces the trees of $\mathcal{T}$ to have a minimum size, it is still worth visiting them.

After pruning, the rest of T_i will be explored by DFS and if an incomplete vertex will be found, we start bDFS with the current remaining tether length for the exploration of *new* edges. The newly explored edges and vertices build a graph G'. If G' has incomplete vertices, we construct a spanning tree T' with a root vertex s', where s' is the vertex in T' closest to s in the current overall explored graph G^*. T' will be inserted into $\mathcal{T}$. After the overall DFS (and bDFS) walk in T_i we delete all trees of $\mathcal{T}$ that are now fully explored. Some of the trees in $\mathcal{T}$ might have common vertices. We merge those trees and build a new spanning tree for them with a new root vertex.

A scheme of the algorithm is shown in Figure 1.31. We have done the prune step by values $(2, 4)$. Otherwise, we have to build very large example graphs.
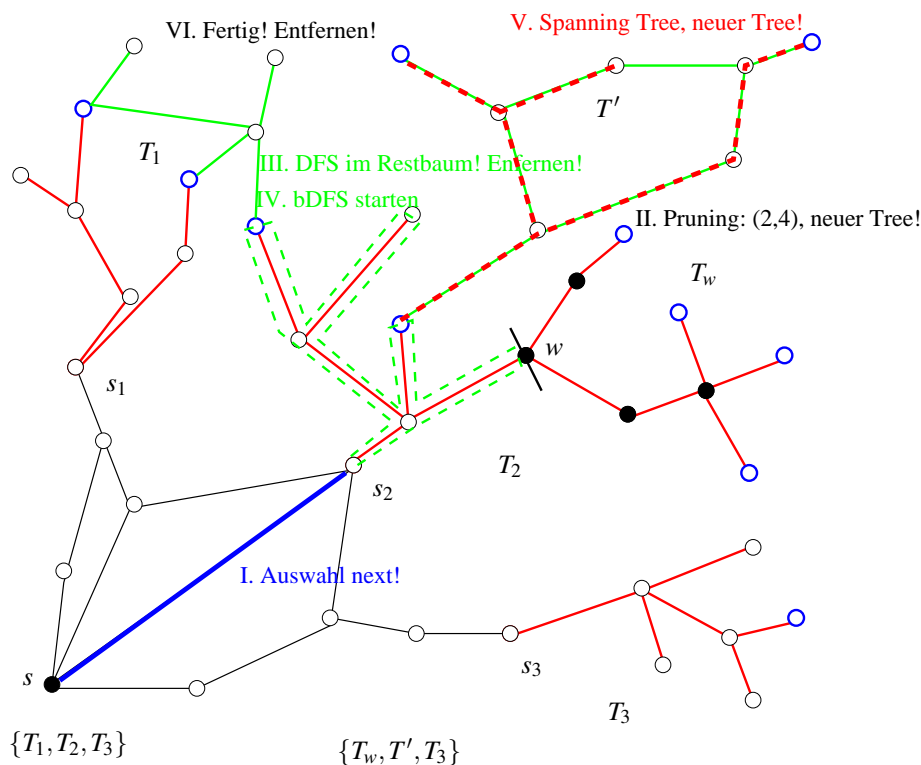
Figure 1.31: The algorithm maintains a set of disjoint trees $\mathcal{T} = \{T_1, T_2, T_3\}$ and choose the tree T_2 with minimal distance $d_{G^*}(s, s_i)$. After that the tree is pruned. Subtrees of distance 2 away from s_2 with vertices inside that have distance at least 4 from s_2 are cut-off. After that DFS starts on the rest of T_2 and starts bDFS on the incomplete vertices. Here some new graphs G' will be explored and we build spanning trees T' for them. Some trees in $\mathcal{T}$ get fully explored. T_w und T' are added to $\mathcal{T}$, the tree T_2 is deleted.

In the following let $d_{G'}(v, w)$ denote the distance between vertices v and w in the subgraph or tree G'. $G^* = (V^*, E^*)$ denotes the currently known part of G.

The algorithm makes us of the following subdivision of vertices:

non-explored a vertex, which was never been visited before.

incomplete a vertex already visited before but some of the adjacent edges are still non-explored.

Algorithm 1.10 CFS

CFS(s, r, α **)**

       $\mathcal{T} := \{\{s\}\}$.
       repeat
         $T_i :=$ closest subtree of $\mathcal{T}$ to s in G^*.
         $s_i :=$ vertex of T_i closest to s n G^*.
         $(T_i, \mathcal{T}_i) :=$ prune(T_i, s_i, $\frac{\alpha r}{4}$, $\frac{\alpha r}{2}$).
         $\mathcal{T} := \mathcal{T} \setminus \{T_i\} \cup \mathcal{T}_i$.
         explore($\mathcal{T}, T_i, s_i, (1 + \alpha)r$).
         Delete fully explored trees from $\mathcal{T}$.
         Merge the trees of $\mathcal{T}$ with common vertices.
         Define a root vertex closest to s in G^*.
       until $\mathcal{T} = \emptyset$

prune(T, v, *minDist*, *minDepth* **)**

       $v :=$ Wurzel von T.
       $\mathcal{T}_i := \emptyset$.
       for all $w \in T$ with $\mathrm{d}_T(v, w) = minDist$ **do**
         $T_w :=$ subtree of T with root w.
         if maximale Distanz between v and a vertex in $T_w > minDepth$ **then**
           // Cut-Off T_w from T ab:
           $T := T \setminus T_w$.
           $\mathcal{T}_i := \mathcal{T}_i \cup \{T_w\}$.
         end if
       end for
       RETURN $(T, \mathcal{T}_i)$

explore($\mathcal{T}$, T, s_i, ℓ **)**

       Move from s to s_i along shortest path in G^*.
       Explore T by DFS. If incomplete vertex occurs, do:
         $\ell' :=$ remaining tether length.
         bDFS(v, ℓ').
         $E' :=$ set of newly explored edges.
         $V' :=$ set of vertices of E'.
         Calculate spanning tree T' for $G' = (V', E')$.
         Define root vertex of T' closest to s in G^*
         $\mathcal{T} := \mathcal{T} \cup \{T'\}$.
       Move back from s_i to s.

---

explored a vertex, that was visited and all adjacent edges have been explored.

Additionally, for the bDFS walk we mark the edges as 'non-explored' or 'explored'.

Lemma 1.23 *The following properties hold during the execution of the CFS–Algorithm:*
 (i) *Any incomplete vertex belongs to a tree in $\mathcal{T}$.*
 (ii) *Until $G^* \neq G$, there is always an incomplete vertex $v \in V^*$ so that $d_{G^*}(s, v) \leq r$.*
 (iii) *For any chosen root vertex s_i: $d_{G^*}(s, s_i) \leq r$.*
 (iv) *After pruning T_i is fully explored by DFS. All trees $T \in \mathcal{T}$ have size $|T| \geq \frac{\alpha r}{4}$.*
 (v) *All trees $T \in \mathcal{T}$ are disjoint (w.r.t. edges).*

Proof.
 (i) Follows directly from the construction of the trees by bDFS and Pruning. No incomplete vertex is missing.
 (ii) Assume that for all $v \in V^*$ we have $d_{G^*}(s, v) > r$ and let v be an incomplete vertex of V^*. In G there is a shortest path $P(s, v)$ from s to v with length $\leq r$. Along $P(s, v)$ there is a first vertex w that does not belong to G^*. Thus its predecessor w' along $P(s, v)$ belongs to V^* and is incomplete. We have $d_{G^*}(s, w') \leq r$.
 (iii) Follows from (ii), the root of a corresponding tree T is always the vertex of T closest to s.
 (iv) We show the property by successively considering the upcoming trees. Or by induction on the number of pruning steps. In the beginning the algorithm starts with bDFS at the root s. Either, the graph will be fully explored and we are done, or bDFS have exhausted the tether of length $(1+\alpha)r$ and have visited more than $(1+\alpha)r$ edges. The single spanning tree T has size $|T| \geq (1+\alpha)r > \frac{\alpha r}{4}$. Let us assume that the condition holds for the trees inside $\mathcal{T}$ and the next pruning step happens. Now by the next iteration we are choosing tree T_i with root s_i closest to s among all trees in $\mathcal{T}$. After that we prune T_i. The rest of T_i has still size $|T_i| \geq \frac{\alpha r}{4}$ since we cut off subtrees T_w with distance $\geq \frac{\alpha r}{2}$ away from s_i. For a corresponding subtree T_w we conclude $|T_w| \geq \frac{\alpha r}{2} - \frac{\alpha r}{4} = \frac{\alpha r}{4}$ since there is a vertex inside T_w that is at least distance $\frac{\alpha r}{2}$ away from s. Now consider the remaining DFS/bDFS combination on (the rest of) T_i. The distance from s to s_i is at most r.
 Any incomplete vertex in the current T_i has at most distance $\frac{\alpha r}{2}$ from s_i otherwise this vertex would be part of a tree T_w that has to be considered in the pruning step. This means that at any incomplete vertex there is a rest tether of length $\frac{\alpha r}{2}$ which can be used for the bDFS part. If the exploration results in another spanning tree T' with incomplete vertices, this tree has size at least $\frac{\alpha r}{2}$.
 Finally fully explored trees are deleted from $\mathcal{T}$ which is not critical. Additionally, some other trees might be merged and still have incomplete vertices. These trees only grow.

$\square$

Finally, we show:

Theorem 1.24 *(Duncan, Kobourov, Kumar, 2001/2006)*
The CFS–Algorithm for the constrained graph-exploration of an unknown graph with known depth is $(4 + \frac{8}{\alpha})$–competitive. [DKK06, DKK01]

Proof. We split the cost for any appearing subtree T_R. Let $K_1(T_R)$ denote the cost for moving from s to s_i in G^*. Let $K_2(T_R)$ denote the cost of DFS for T_R and let $K_3(T_R)$ denote the cost for the bDFS exploration done for the incomplete vertices starting at T_R. The trees are edge disjoint.

The total cost is a sum of the cost for any T_R. We have

$\sum_{T_R} K_3(T_R) \leq 2 \cdot |E|$, since bDFS only visits non-explored edges (twice).

$\sum_{T_R} K_2(T_R) = \sum_{T_R} 2 \cdot |T_R| \leq 2 \cdot |E|$, the cost for all DFS walks.

For $K_1(T_R)$ we have $K_1(T_R) = 2 \cdot d_{G^*}(s, s_i) \leq 2r$. The complexity of any T_R is at least $\frac{\alpha r}{4}$ which gives $|T_R| \geq \frac{\alpha r}{4}$ for the number of edges. We conclude $r \leq \frac{4|T_R|}{\alpha}$ and

$$\sum_{T_R} K_1(T_R) \leq \sum_{T_R} 2r \leq \frac{8}{\alpha} \sum_{T_R} |T_R| \leq \frac{8}{\alpha} |E|$$

Altogether, the algorithm makes $(4 + \frac{8}{\alpha})|E|$ step whereas any optimal algorithm visits at least any edge once.   □

In general we assume that α is a small constant with $0 < \alpha < 1$. The above proof works for any $\alpha > 0$. The cost of the algorithm for known depth r are within $O(|E|/\alpha)$. More precisely we can show that actually $O(|E| + |V|/\alpha)$ steps are made. For this we have a closer look at the cost. bDFS work on the edges only. The DFS walk work on trees where the number of vertices is the same as the number of edges. Some of these vertices appear in two trees, so by a factor of 2 we are on the save side. The movements from s to s_i are analysed over the size of spanning trees, where vertices and edges are also the same.

The cost $K_1(T_R)$ and $K_2(T_R)$ sum up to $\left(2 + \frac{8}{\alpha}\right) 2|V|$.

Altogether there is an $\Theta(|E| + |V|/\alpha)$ algorithm for the exploration of arbitrary graphs.

Corollary 1.25 *The CFS–Algorithm for the constrained graph-exploration of an unknown graph with known depth has optimal exploration cost $\Theta(|E| + |V|/\alpha)$.*

Now we have some possibilities for extensions. First, we assume that the depth of the graph is unknown in the beginning. Next we would like to consider weighted edges.

1.5.1   Restricted graph-exploration with unknown depth

Let is now assume that the radius, say R, of the unknown graph G is not known. From a practical point of view, spending some cable is costly and we would like to extend the tether only if it is necessary. A first simple idea is that we guess the depth, say r, and successively double its length until the algorithm finally explores the whole graph. Obviously, the repeated application of the CFS-algorithm runs in $O(\log r |E|)$ step. As shown above we can also refine the analysis of this approach. For any bDFS step we make use of the already visited edges and directly *jump* to incomplete vertices (now with larger tether length). Therefore the bDFS steps are still subsumed by $2|E|$ steps. But we still have to take the movements to the roots of the trees into account as well as the DFS movements on the new subtrees. Therefore we have the following result.

Corollary 1.26 *Applying the CFS–Algorithmus by successively doubling the current depth r gives an algorithm that explores an unknown graph G with unknown depth R with $\Theta(|E| + (\log R)|V|/\alpha)$ steps.*

We will now show that we can get rid of the log-factor by successively adjusting r appropriately. We only exchange two calls in the main procedure. In principle, instead of the known value r we successively make use of $r := d_{G^*}(s, s_i)$, which is the smallest distance from s to one of the roots of the trees in $\mathcal{T}$.

More precisely, we exchange prune(T_i, s_i, $\frac{\alpha r}{4}$, $\frac{\alpha r}{2}$) wird by prune(T_i, s_i, $\frac{\alpha d_{G^*}(s,s_i)}{4}$, $\frac{9\alpha d_{G^*}(s,s_i)}{16}$) and explore($\mathcal{T}$, T_i, s_i, $(1 + \alpha)r$) by explore($\mathcal{T}$, T_i, s_i, $(1 + \alpha)d_{G^*}(s,s_i)$). This means that the pruning-step is done with the values $\frac{\alpha d_{G^*}(s,s_i)}{4}$ and $\frac{9\alpha d_{G^*}(s,s_i)}{16}$ and the eplore-step is done with tether length $(1 + \alpha)d_{G^*}(s,s_i)$.

In the beginning we have $d_{G^*}(s, s_i) = 0$, therefore we make use of some fixed constant c in the beginning and use $r := max(d_{G^*}(s,s_i), c)$. Let $d_{G^*}(s, T)$ denote the shortest distance from s to some vertex in T inside G^*.

Lemma 1.27 *For the CFS–Algorithmus with unknown depth R we have the following properties:*
 (i)   Any incomplete vertex belongs to a tree in $\mathcal{T}$.
 (ii) There is always an incomplete vertex $v \in V^*$ with $\mathrm{d}_{G^*}(s,v) \leq r$, until $G^* \neq G$.
 (iii) For the closest root s_i we have: $\mathrm{d}_{G^*}(s,s_i) \leq r$.
 (iv) For all trees $T \in \mathcal{T}$ we have $|T| \geq \frac{\max(d_{G^*}(s,T),c)\alpha}{4}$. After pruning the remaining tree will be fully explored by DFS.
 (v)   All trees ever considered in $\mathcal{T}$ are (edge) disjoint.

Proof. For the proof of (i),(ii),(iii) and (v) we apply the same arguments as in the proof of Lemma 1.23. It remains to show that (iv) holds. The main difference is that the size of a tree T is directly correlated to the distance from s to T, this is different from the previous argumentation.

Let us first show that the remaining tree T_i (after pruning) will be fully explored by DFS. For any vertex v in T_i we have $d_{T_i}(s_i, v) \leq \frac{9 d_{G^*}(s,s_i)\alpha}{16}$, otherwise v has been cut of by pruning. Thus we have

$$(1+\alpha)d_{G^*}(s,s_i) - d_{G^*}(s,s_i) - d_{T_i}(s_i,v) \geq \frac{7 d_{G^*}(s,s_i)\alpha}{16},$$

which shows that the tether is long enough T_i will be fully explored by DFS.

By induction over the number of pruning steps we will finally show: $\forall T \in \mathcal{T} : |T| \geq \frac{\max(d_{G^*}(s,T),c)\alpha}{4}$.

In the beginning we apply bDFS from the start with tether length c. Either we explore the whole graph or we have $|T| \geq (1+\alpha)c > \frac{\alpha c}{4}$ for the resulting spanning tree T. For simplicity we assume $d_{G^*}(s,T_i) > c$ from now on.

We would like to show that for any tree T_w, resulting from the pruning of some T_i, we have $|T_w| \geq \frac{d_{G^*}(s,T_w)\alpha}{4}$. Also the remaining tree T_i has this property.

For the remaining tree T_i (after pruning), we conclude $d_{G^*}(s,T_i) = d_{G^*}(s,s_i)$ and pruning guarantees $|T| \geq \frac{d_{G^*}(s,T)\alpha}{4}$. For a tree T_w pruned from T_i we have: $|T_w| \geq \frac{9 d_{G^*}(s,s_i)\alpha}{16} - \frac{d_{G^*}(s,s_i)\alpha}{4} = 5\frac{d_{G^*}(s,s_i)\alpha}{16}$ by the pruning values. Additionally, we have $d_{G^*}(s,T_w) \leq d_{G^*}(s,s_i) + d_{G^*}(s_i,w) = (1+\frac{\alpha}{4})d_{G^*}(s,s_i)$, since the root w of T_w is exactly $\frac{\alpha d_{G^*}(s,s_i)}{4}$ steps away from s. Für $0 < \alpha < 1$ we conclude: $d_{G^*}(s,T_w) < \frac{5 d_{G^*}(s,s_i)}{4}$ and together with the above inequality we have $|T_w| > \frac{d_{G^*}(s,T_w)\alpha}{4}$.

Finally, we have to analyse the emerging spanning trees T_v, which will be constructed from the bDFS steps starting during the DFS walk in T_i. Such a tree T_v *starts* at some incomplete vertex v in T_i. We have $d_{G^*}(s_i,v) \leq \frac{9 d_{G^*}(s,s_i)}{16}$, otherwise v would have been pruned and could not be a leaf of the rest of T_i any more. Thus we have $d_{G^*}(s,T_v) \leq d_{G^*}(s,s_i) + d_{G^*}(s_i,v) < \frac{25 d_{G^*}(s,s_i)}{16}$ or $d_{G^*}(s,s_i) > \frac{16 d_{G^*}(s,T')}{25}$. If T_v is fully explored, we are done, since the tree will be deleted. Assume that T_v still has incomplete vertices. As mentioned above we have $d_T(s_i,v) \leq \frac{9\alpha d_{G^*}(s,s_i)}{16}$. Starting from v there was a remaining tether length of $\frac{7\alpha d_{G^*}(s,s_i)}{16}$ for the construction of the incomplete T_v, which gives $|T_v| \geq \frac{7\alpha d_{G^*}(s,s_i)}{16}$. Application of $d_{G^*}(s,s_i) > \frac{16 d_{G^*}(s,T_v)}{25}$ gives $|T_v| > \frac{7\alpha d_{G^*}(s,T_v)}{25} > \frac{d_{G^*}(s,T_v)\alpha}{4}$. Either we have explored everything behind v or the spanning tree T_v has size $|T_v| > \frac{d_{G^*}(s,T_v)\alpha}{4}$.

We have considered any emerging $T \in \mathcal{T}$! $\square$

Theorem 1.28 *(Duncan, Kobourov, Kumar, 2001/2006)*
Applying the CFS–Algorithm with the adjustments above results in a correct restricted graph-exploration of an unknown graph with unknown depth. The algorithm is $(4+\frac{8}{\alpha})$–competitive.   *[DKK06, DKK01]*

Proof. We apply the same analysis as in the proof of Theorem 1.24. For the analysis of the movements from s to the roots of the trees we make use of the correlation $|T_R| > \frac{d_{G^*}(s,T_R)\alpha}{4}$. $\square$

For the number of steps we can also refine the analysis, analogously.

Corollary 1.29 *The above CFS–Algorithm for the restricted exploration of an unknown graph with unknown depth requires $\Theta(|E|+|V|/\alpha)$ exploration steps, which is optimal.*

Finally, we would like to argue that the usage of a look-ahead of αr is necessary for attaining linear optimal exploration cost (i.e., in comparison to $|E|$ and $|V|$. This can be shown for the accumulator variant as follows. First, it is clear that an accumulator of size $2r$ is not sufficient for exploring all edges. The graph in Figure 1.32 has depth 6, but exploring all edges requires an accumulator of size 13.

This means that an accumulator size $2r+1$ is necessary. We show that an accumulator of size $2r+d$ for constant d is not sufficient in the sense of performing no more than $C \cdot |E|$ exploration steps.

Lemma 1.30 *For the accumulator variant with accumulator size $2r+d$ for constant d, there are examples do that any algorithm attains at least $\Omega\left(|E|^{\frac{3}{2}}\right)$ exploration steps.*
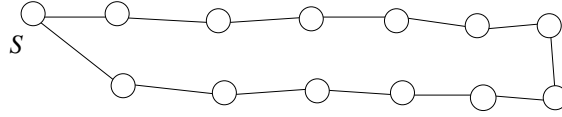
Figure 1.32: A graph of depth $r = 6$ that cannot be explored by an accumulator of size $2r$.

Proof. We consider the following example as given in Figure 1.33. Starting from s there is a path of length $\frac{n}{2}$ that visits a clique of size $\frac{n}{2} + 1$. Moving forth and back along the path requires n steps, the depth of the graph is $\frac{n}{2} + 1$. Exploration with accumulator size $n + 2 + d$ means that we have to visit the clique $\Omega\left(\frac{n^2}{d}\right)$ times since the clique has $\Omega\left(n^2\right)$ edges. This gives $\Omega\left(\frac{n^2}{d} \cdot n\right) = \Omega\left(n^3\right)$ exploration steps. The statement follows from $|E| \in \Theta(n^2)$.   □
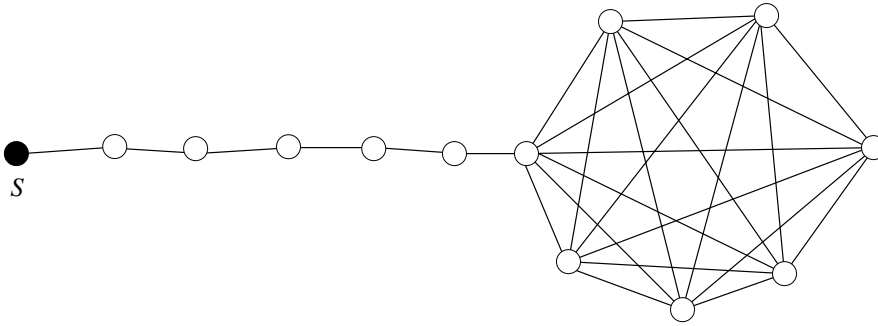

Figure 1.33: A graph with $n + 1 = 13$ vertices. A path of length $\frac{n}{2}$ visits a clique of size $\frac{n}{2} + 1$. Any accumulator strategy with accumulator size $n + 2 + d$ requires $\Omega(n^3)$ steps.

With a similar argument we conclude that an sub-linear extension of the accumulator, i.e., size $2r + o(r)$, is not sufficient for attaining a linear cost strategy. Let us briefly repeat the small-o notation. For real valued functions or series f and g we define $f \in o(g)$, if and only if $\lim_{r \to \infty} \frac{f(r)}{g(r)} \to 0$ holds. Therefore we conclude $r \in o(r^2)$, $c \in o(r)$ for any constant c and also $\frac{1}{r} \in o(1)$. By the above arguments and example we can show that $\Omega\left(\frac{n^3}{f(n)}\right)$ exploration steps are necessary for an accumulator of size $n + 2 + f(n)$. For $f(n) = n^{1-\varepsilon}$ (this means $f \in o(n)$) we have to perform $\Omega(|E|^{1+\varepsilon})$ exploration steps.

Note, that for the tether variant up to our knowledge there is no such statement that a tether of length $r + o(r)$ is necessary for attaining $O(|E|)$ exploration cost.

We have shown that we can explore any graph (online and offline) with at most $\Theta(|V| + |E|)$ exploration steps. These are the pure cost for the motion of the agent. In the literature this is also denoted as the *mechanical cost*; see also [DJMW91]. Besides, there are also some *computational cost*, for the planning and preparation of the strategy.

For example the computational cost of the CFS-Algorithm have to be analysed for the following tasks:

- Build the spanning trees

- Update the shortest paths to the trees of $\mathcal{T}$

- Merge the trees

- Detect fully explored trees

- Prune a tree

- Maintain the list $\mathcal{T}$

- Apply DFS/bDFS

For unit-lenght edges some of the above tasks can be done very efficiently. The overall approach can be easily extended to weighted graphs (positive edge weights).

Exercise 10 *Analyse the computational cost for the CFS-Algorithm in $O-$notation for $|E|$ and/or $|V|$.*

Exercise 11 *Show that the CFS-Algorithm approach also works for graphs with positive edges weights. How do we have to adjust the CFS-Algorithm?*

1.5.2 Mapping of an unknown graph

Finally, in this section we would like to show the influence of different capabilities of the agent. Up to now we assumed that an already visited vertex or edge will be recognized at the next visit. This means that we have marked any visited edge and vertex.

Let us now assume that the agent cannot mark parts of the environment. We do not have any *landmarks*. We still assume that we have enough storage for constructing the sub-graph detected so far.

The following model is taken from Dudek et al.; see[DJMW91]. The agent has no orientation and no compass. At any vertex the outgoing edges are presented in the same order. This order need not represent a planar embedding. If the agents visits the vertex from different incoming edges, the order will be consistent. This means that there is a fixed cyclic order, the relative presentation of the order stems from the edge where the agent currently comes from. Figure **??** shows an example of a relative order. By this order, the agent knows where he was coming from and can also return to this vertex. Since the storage is not limited, it is possible to remember a return path. Let us for example assume that the agent visits vertex v_2 by edge e_1 and then visits the second edge e_3 in ccw-order from e_1. If the agent moves back along e_3 to v_2, it already knows that it was recently coming from the first edge in ccw order, which is e_1. The agent can make use of this return path. If the agent visits a vertex in a forward step, it has no idea which of the vertices the visited vertex actually is.

Is it possible to build a map of the graph and to locate oneself inside the graph? The offline input is a triple $G = (V, E, S)$, where by S for any vertex the cyclic local order of the edges is given.

First, it is easy to see that without further capabilities, one can not fully detect a given graph. Figure 1.34 shows two different regular graphs of fixed degree 3. For an agent the information on any vertex is exactly the same. It is not possible to distinguish between the two variants. At least one marker is necessary.
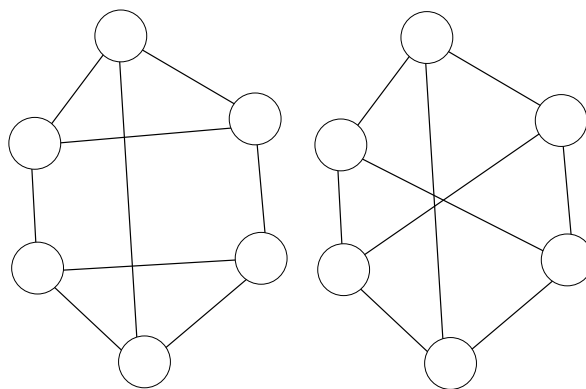

Figure 1.34: Two different regular graphs of degree 3, an agent cannot distinguish them without a marker.

Corollary 1.31 *Let $G = (E, V, S)$ be a graph with local cyclic edge order- Without a marker an online agent cannot build a correct map of the graph.*

Exercise 12 *Give a formal argument that the graphs in Figure 1.34 are different. Which class of graphs can be correctly detected by an online agent without a marker?*

A single marker (or pebble) is sufficient as shown by Dudek et al. [DJMW91]. We describe the corresponding *Marker*-Algorithm. The algorithm maintains the current known graph G^* and a list L of non-determined (seen but not correctly detected) edges. In the beginning the starting vertex is known and its outgoing edges belong to L. They are given in the cyclic order.

In the main step, the algorithm choose an edge e of L starting at a detected vertex b and moves to a vertex u along the edges $e = (b, u)$. Now the agent sets the marker on u, moves back to b along e and searches for the pebble in G^*.

Case 1: The pebble was not found in G^*. In this case we add the edge $e = (b, u)$ to G^* w.r.t. the cyclic order. All outgoing edges of u different from e will be inserted into the list L of non-determined edges.

Case 2: The marker has been found at some vertex $v \in G^*$. If there is more than one non-determined outgoing edges at $v = u$, we cannot precisely detect e. Therefore we take the marker, move back to b, place the pebble there, move back to v again and successively check the non-determined edges. Finally, we will detect the edge e and add it to G^* by the local order.

The above algorithm is simple and correct. By construction in any step an additional edge will be correctly detected. The number of exploration steps is restricted by $O(|E| \times |V|)$ and the same holds for the computational cost. We assume that the graph is not a multigraph and has no loop edges (v, v). Besides, we assumed that any edge has unit-length.

Theorem 1.32 *(Dudek, Jenkin, Milios, Wilkes, 1991)*
Let $G = (E, V, S)$ be a graph with given cyclic local order of the edges. By the use of one marker it is possible to fully detect the structure of the graph by online navigation with $O(|E| \times |V|)$ exploration steps and also overall $O(|E| \times |V|)$ computational cost.

Proof.
Let $G^* = (V^*, E^*, S^*)$ be the current graph during the execution of the Marker-Algorithm. Setting the marker has cost $O(1)$, searching for the marker in G^* can be done by DFS by $O(|V^*|)$ steps. Moving back and force along a path can be done in $O(|V^*|)$ steps as well. The traversal cost are considered for any edge, which gives $O(|E| \times |V|)$ steps in total.

For unit-edge length the computational cost are precisely the same for any edges we have to compute the shortest paths between two vertices. The effort is bounded by $O(|V^*|)$. This gives $O(|E| \times |V|)$. $\quad\square$

Exercise 13 *Explain why the cyclic order of the edges is necessary for the above Marker-Algorithm. Where is it used during the execution of the algorithm?*

Exercise 14 *Analyse the mechanical and the compuational cost of the marker algorithm for graphs with positive edge weights.*

Chapter 2

Polygonal enviroments

In this section we turn over to planar environments modelled by (a set of) simple polygons. We assume that a set of simple polygons P_i for $i = 1, \dots, k$ is given. Two polygons do neither intersect nor touch each other. The number of polygons is finite in the sense that any circle of fixed radius contains only finitely many *obstacles* P_i.

In the following sections an agent tries to escape from a labyrinth (modelled by polygons) or tries to find a target t in a polygonal environment. We assume that the agent is point-shaped and thus consider curves in the plane. From a practical point of view one might think that a physical robot somehow *follows* the corresponding curve without actually running precisely on it. Additionally, in the followning configurations, the agent has only a limited storage.

We make use of the following conventions. If the coordinates of the target are given, the task of the agent is denoted as "Navigation". On the other hand, if the coordinates are not known the task of the agent is "Searching". We will consider different sensor models.

Some of the following algorithms – for example the Plegde-algorithm and some Bug-Variants – have been implemented as interactive Java-Applets, see

    http://web.informatik.uni-bonn.de/I/GeomLab/Polyrobot/

2.1 Escape from the labyrinth

The task of the agent is to escape from a polygonal environment. In the geometric sense the agent escapes, if it finally hits a circle that contains all obstacles.

The agent is point-shaped and makes use of a touch-sensor for following the wall by Left-Hand-Rule or Right-Hand-Rule. Additionally, we allow that the agent can count its total turning angles in a single ; see Figure 2.2(i). The agent realizes when it hits the enclosing circle. In this case the agent successfully escaped from the labyrinth.

2.1.1 Pledge-Algorithm

Algorithm 2.1 Pledge-Algorithmu

1. Choose direction φ, turn agent into direction φ.

2. Move into dircetion φ, until an obstacle is met.

3. Turn right and follow the wall by Left-Hand-Rule.

4. Follow the wall, sum up the overall turning angles, until the angular counter gets zero, in this case GOTO (2).

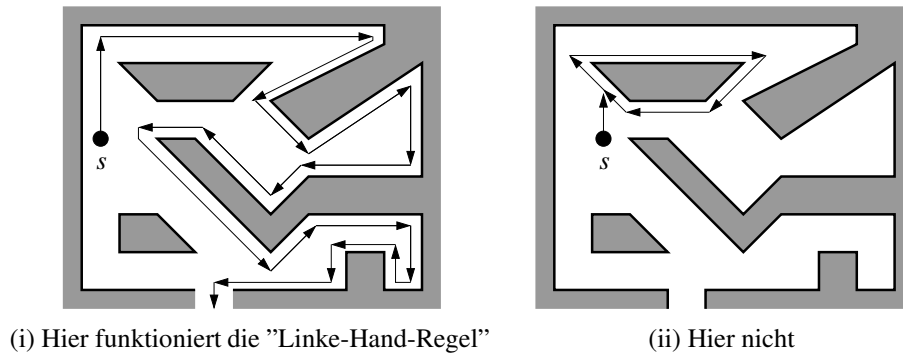(i) Hier funktioniert die "Linke-Hand-Regel"   (ii) Hier nicht

Figure 2.1: Simple strategies cannot be successful.

Note that, simple counting schemes or movements cannot succeed. For example, following the wall by Left-Hand-Rule until the agent points again into direction φ and leave the obstacle right now can result in infinite loops; see Figure 2.2(ii). Just following the boundary could let the agent stuck inside the labyrinth; see Figure 2.1.
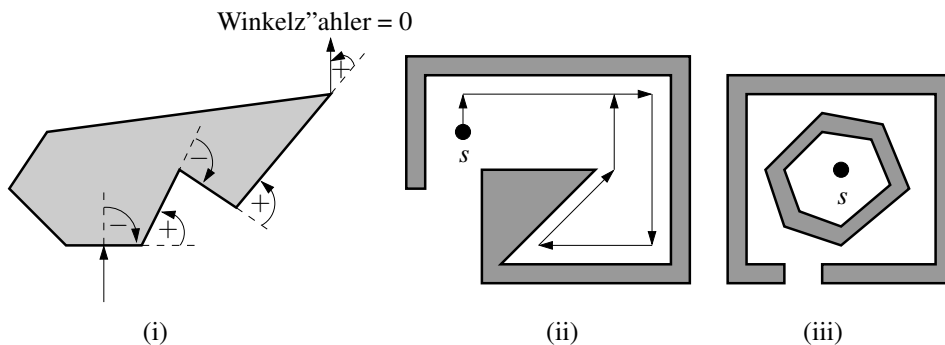
Figure 2.2: (i) Angular counter. (ii) Leave-condition "angular counter mod $2\pi = 0$" is not sufficient. (iii) The agent cannot escape.

The following simpe Pledge-Algorithm (Algorithm 2.1) solves the problem. For the correctness proof we require structural properties.

Lemma 2.1 *The angular counter of the Pledge-Algorithm never attains a positive value.*

Proof.
- The angular counter is initialized by zero.
- After hitting an obstacle, the counter gets negative by the first right turn.
- The counter changes continuously, the agent leaves the obstacle if the counter gets zero again.
$\Rightarrow$ the statement is true.

$\square$

Lemma 2.2 *If the agent does not leave the labyrinth by the Pledge-Algorithm, at the end there will be a finite path $\pi_\circ$, so that the agent follows this path infinitely often.*

Proof. The path generated by the Pledge-Algorithm is a polygonal chain. The set of vertices S stem from the vertex set of all obstacles plus vertices inside edges that occur when the algorithm leaves an obstacle at a vertex and hits an edges afterwards. The algorithm leaves an obstacle only at a vertex. Thus the overall number of possible vertices is bounded. If the Pledge leaves an obstacle at the same vertex twice, after that the same path $\pi_\circ$ will be used again and again, since the algorithm is deterministic and we left the same vertex with the same counter twice.

Therefore, if such path does not exists and the agent is not successful, finally the path will not leave an obstacle anymore. Obviously, the path $\pi_\circ$ exists in this case. □

Lemma 2.3 *Assume that the agent does not leave the labyrinth by the Pledge-Algorithm, the above mentioned finite path $\pi_\circ$ does not have self-intersections.*

Proof. An intersection can only occur at the boundary, since all movements (segments) in the *free space* are arranged in parallel. [1]

Let us assume that a part B of $\pi_\circ$ intersects with a former path A of $\pi_\circ$ at the boundary of some P_i. This means that there is a point $z \in P_i$ where B (running from the free space) hits A for the first time. The corresponding segment $e \in P_i$ points into a fixed direction. Therefore after turning, the turning angle, $C_B(z')$, of B at some point $z' \in e$ closely behind z and the turning angle, $C_A(z')$, differ by $2k\pi$ for some $k \in \mathbb{Z}$. If e cause turning angle $-\beta$ for B at z, we have $C_B(z') = -\beta$ and $C_A(z') = -\beta + 2k\pi$ with $k \in \mathbb{Z}$. For $k > 0$ we have:

$$C_A(z') = -\beta + 2k\pi > -\pi + 2\pi = \pi \nleftrightarrow \text{by Lemma 2.1.}$$

For $k = 0$ it is clear that $C_A(z') = C_B(z')$ and A and B will never diverge, this contradicts the given situation that A and B are parts of $\pi_\circ$.

From $k < 0$ we conclude $C_A(p) < C_B(p)$ for all points p between z' and the first point z'', where the paths of A and B separate. From $C_A(p) < C_B(p)$ we conclude that at z'' the path B has $C_B(z'') = 0$ and leaves the obstacle. □

Finally, we conclude:

Theorem 2.4 *(Abelson, diSessa)*
For any given polygonal labyrinth, the Pledge-Algorithm will be able to escape from the labyrinth from any starting point, from which a successful path exists. *[Ad80]*

Proof. Assume, that the agent is not successful in the given situation. By Lemma 2.2 follows the finite path $\pi_\circ$ again and again, and the path $\pi_\circ$ has no self-intersections. Either the agent runs along $\pi_\circ$ in ccw-order or in cw-order. In ccw-order the angular counter will increase by $+2\pi$ in any round, which contradicts Lemma 2.1. In cw-order the angular counter will decrease by -2π in any round, thus at some point, the agent cannot leave an obstacle any more and in turn $\pi_\circ$ already belongs to a single obstacle. The agent follows the wall of an obstacle by Left-Hand-Rule and in cw-order, this can only mean that the agent is enclosed by the obstacle; see Figure 2.2(iii). There is no way out of the labyrinth from the given starting point. □

2.1.2 Pledge-Algorithm with sensor errors

The correctness proof of the Pledge-Algorithm (Theorem 2.4) make use of the assumptions that a point-shaped agent counts the angles without any errors and moves precisely into a specified starting direction. setzt voraus, dass der Roboter punktf''ormig ist und fehlerfrei arbeitet. As already mentioned physically the agent need not be a point, we can assume that the agent requires some room for its movement and in principle follows a curve calculated by the Pledge-Algorithm. The main problem is that the curve itself is not computed exactly by the robots abilities. For example the agent cannot precisely measure the turning angles at the boundary and cannot precisely follow a direction.

If the agent makes gross faults, we cannot assume that the pledge-like behaviour will succeed. Is there an error bound for the sensors that still allows to escape by a pledge-like behaviour?

---

[1] All segments that are not part of the boundary of some P_i.

The general idea is as follows. We would like to design a class of curves $\mathcal{K}$ of the configuration space. Such a curve will be computed by an agent with sensor errors and imprecise measurements. Any curve from $\mathcal{K}$ represents a possible path for the escape. As mentioned above, the agent is guided by the computed curve and moves close to it. We would like to define sufficient conditions for the curves, such that the escape is guaranteed, if a corresponding path exist. The curve always have precise orientation. Its computation might be erroneous.

As the curves guides the agent, for convenience, we can consider curves in the configuration space by $C(t) = (P(t), \varphi(t))$ where $P(t) = (X(t), Y(t))$ is the location in the work space and $\varphi(t)$ the current turning angle. Note that two full turns will result in a turning angle of 4π instead of zero. Therefore $\varphi(t) \in \mathbf{R}$! A configuration (x, y, φ) is **half-free**, if the curve touches an obstacle in the work space and **free**, if no obstacle in the work space is met. The set of all half-free points is denoted by $\mathcal{C}_{\text{half}}$ whereas the set of all free points is denoted by $\mathcal{C}_{\text{free}}$. Points, where the curve moves from $\mathcal{C}_{\text{free}}$ to $\mathcal{C}_{\text{half}}$ are called Hit-Points, h_i. Points, where the curve leaves $\mathcal{C}_{\text{half}}$ and enters $\mathcal{C}_{\text{free}}$ are denoted as Leave-Points, ℓ_i. For simplicity, we will also denote the corresponding time parameter by h_i or ℓ_i, respectively.

The Pledge-Algorithm has two movement modi. Either the agent follows the wall and counts turning angles or the agent moves in the free space. Both movements can be erroneous, the agent diverges from the starting direction in the free space and drifts off or the agent cannot count turning angles precisely and will leave the obstacle earlier or later than in the original pledge path.

In principle we have to avoid that the agent moves in infinite loops. The figure shows that a large drift can easily result in a loop. The error of the agent is too large. But also a small deviation in the free space after each leave-point can sum up to a large total drift and an infinite loop. Figure 2.3 shows an example where there are small drifts after each leave that finally results in an infinite loop. This means that the direction in the free space should be globally stable.
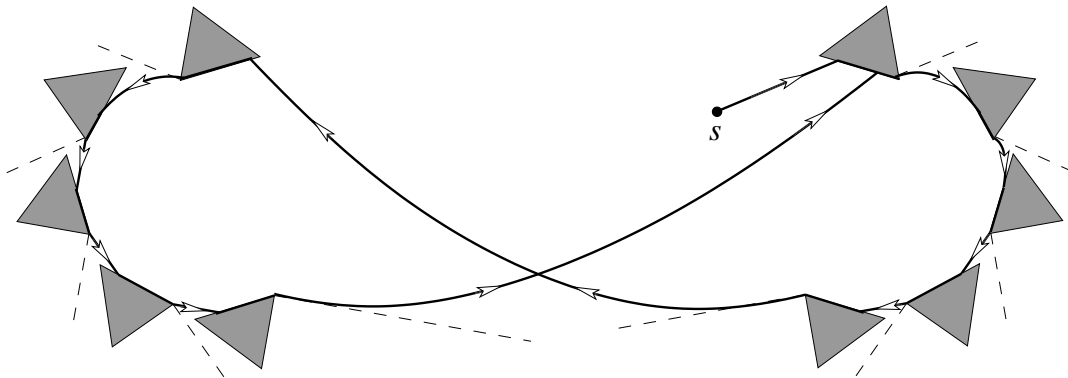

Figure 2.3: Small deviations sum up to a large deviation.

Additionally, for leaving the environment it is highly recommended that the agent at least moves into a certain global direction. One might also think that an erroneous compass will at least allow us to move generally into a half-plane. Therefore we require a $\mathcal{C}_{\text{free}}$-condition:

$$\forall t_1, t_2 \in C : P(t_1), P(t_2) \in \mathcal{C}_{\text{free}} \Rightarrow |\varphi(t_1) - \varphi(t_2)| < \pi$$

which guarantees that the angular counter in the free space maximally differs from the starting direction $\varphi(s)$ only by a fixed value. If we make use of a compass it seems to be reasonable to think that we can guarantee $\forall t : \varphi(t) \in]-\frac{\pi}{2}, +\frac{\pi}{2}[$ in the free space for starting direction zero.
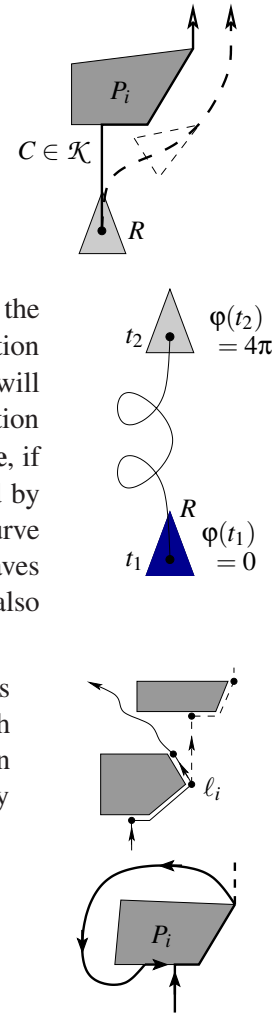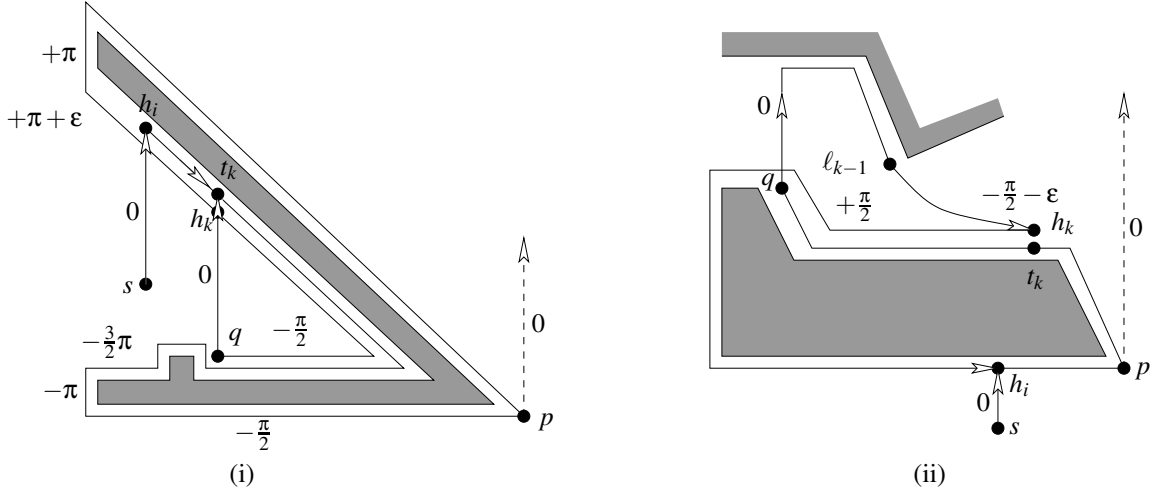
Figure 2.4: A local overturn of the angular counter can result in infinite loops.

Unfortunately, the C_{free}-condition is not sufficient. We have to combine it with the angular counter at the obstacles. Figure 2.4 shows two examples where the agent overturns the counter at the obstacle for a while (because of sensor errors) but obtains overall precise values later for the leaving condition. This has nothing to do with the free space condition. The infinite loop happens at the obstacle. In this case the agent passes the first true leave-point p and leaves the obstacle later at a point q which is also a legal leave-point.

In Figure 2.4(i) by the overturn the curve mets the same obstacle h_k at again. In Figure 2.4(ii) by the overturn the agent first visit another obstacle but then returns via some leave-point ℓ_{k-1} again to the starting obstacle at h_k. In both case $P(h_k)$ is visited twice and during the first visit at t_k the angular counter was overturned. In the left hand side figure the angular counter at t_k is larger than π, in the right hand side figure the counter at t_k is $+\frac{\pi}{2}$. Figure 2.4(ii) shows a second error source. Since the angular counter at h_k is $-\frac{\pi}{2} - \varepsilon$ both errors also sum up to an error larger than π. We state that for the hit-point h_k the angular counter of a previous visit was overturned. Together with the error at h_k there is an overall overturn larger than π.

We subsume the requirement in the C_{half}-condition:

$$\forall h_i, t \in C : P(t) = P(h_i) \Rightarrow \varphi(t) - \varphi(h_i) < \pi.$$

We can also think about the compass with a deviation of $\pi/2$ into both directions. This means that we can overturn the angle counter at the obstacle by less than $\pi/2$. Together with a deviation of less than $-\pi/2$ in the free space we might hit this point again but $\varphi(t) - \varphi(h_i) < \pi$ holds.

Definition 2.5   Let $\mathcal{K}$ be a class of curves in $C_{\text{free}} \cup C_{\text{half}}$ such that any curve from $\mathcal{K}$ fulfils the following conditions:
  (i) Parameterized pledge like curve with turn-angles and position:
       $C(t) = (P(t), \varphi(t))$ with $P(t) = (X(t), Y(t))$
 (ii) At the boundary the curve surrounds obstacles by Left-Hand-Rule.
(iii) Any leave-point is a vertex of an obstacle.
 (iv) C_{free}-condition holds: $\forall t_1, t_2 \in C : P(t_1), P(t_2) \in C_{\text{free}} \Rightarrow |\varphi(t_1) - \varphi(t_2)| < \pi$
  (v) C_{half}-condition holds: $\forall h_i, t \in C : P(t) = P(h_i) \Rightarrow \varphi(t) - \varphi(h_i) < \pi$

Obviously, the path constructed by the error-free Pledge-Algorithm does belong to $\mathcal{K}$. For the correctness proof that any curve of $\mathcal{K}$ that is constructed in a pledge-like fashion will escape from the labyrinth. We show some important structural properties of curves in $\mathcal{K}$.

Lemma 2.6   *A curve $C \in \mathcal{K}$ has no self-intersection.*
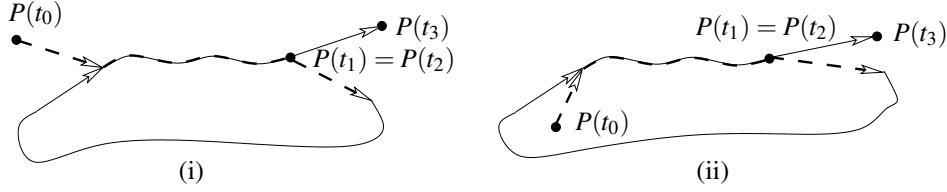
Figure 2.5: The difference between (i) crossing and (ii) touching at t_2.

Note that a curve from $\mathcal{K}$ might touch itself; see Figure 2.5.

Proof. Assume that C has an intersection, consider the first intersection. There are parameter t_1 and t_2 with $t_1 < t_2$ and $P(t_1) = P(t_2)$ and the first intersection occurs at $P(t_2)$.

This means that between t_1 and t_2 there is a cw or ccw turn. If the intersection lies in the free space, obviously the $\mathcal{C}_{\text{free}}$-condition does not hold. Thus we can assume that $P(t_2)$ is in $\mathcal{C}_{\text{half}}$ holds. Consider the case of a cw loop as depicted in Figure 2.6.
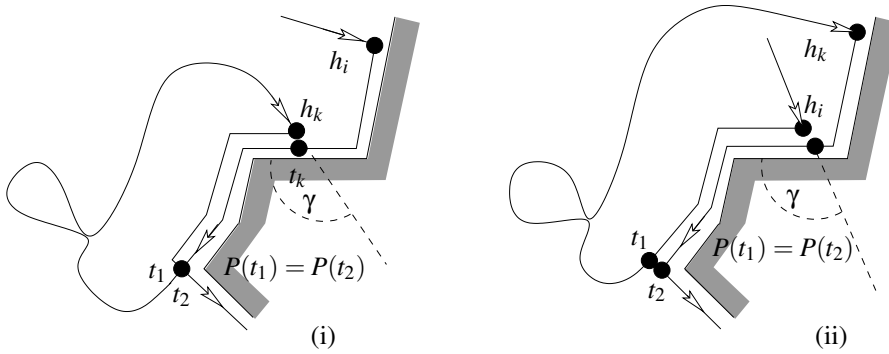

Figure 2.6: Cw loop and two cases.

The curve hits the obstacle at h_i, follows the wall, leaves the obstacle at time t_2 comes back to the obstacles at $h_k > h_i$ and has an intersection at t_2; see Figure 2.6(i). If the point $P(h_k)$ was not visited between h_i and t_1 there is only a touching event at t_2; see Figure 2.6(ii).

Let $\varphi(h_k^+)$ deonte the angular counter after the agent turns into the direction of the corresponding edge. we have $\varphi(h_k^+) = \varphi(h_k) + \gamma$, where γ denotes the turning angle at the edge. We have $-\pi < \gamma < 0$. Additionally, by the full cw turn we conclude $\varphi(h_k^+) = \varphi(t_k) - 2\pi$. Also the $\mathcal{C}_{\text{half}}$-holds and we obtain:

$$\varphi(t_k) - \varphi(h_k) < \pi$$
$$\Leftrightarrow \quad \varphi(h_k^+) + 2\pi - \varphi(h_k) = \varphi(h_k) + \gamma + 2\pi - \varphi(h_k) < \pi$$
$$\Leftrightarrow \quad \gamma < -\pi \; \lightning$$

Similar arguments hold for a loop in ccw order, which is an exercise. The first intersection cannot exists. By induction there is no intersection at all. $\square$

Lemma 2.7 *A curve $C \in \mathcal{K}$ hits any edge of the environments at most once.*

Proof. Assume that a single edge e has two hit-points. After the first hit h_i of edge e the curve can only leave the obstacle at a vertex and then comes back to e at h_k; see Figure 2.7.

In $P(h_i)$ and $P(h_k)$ the agent turns clockwise in order to follow the edge e which gives turning angles $-\pi < \gamma_i, \gamma_k < 0$. Let $\varphi(h_i^+)$ and $\varphi(h_k^+)$ denote the turning angles as in the proof of Lemma 2.6 (turning angle directly after the hit-point). Let w. l. o. g. $\varphi(h_i^+) = 0$.

In h_i^+ and h_k^+ the curve follow the edge e, and the direction differs only by $2j\pi$ for some $j \in \mathbb{Z}$ This means $\varphi(h_k^+) = 2j\pi, j \in \mathbb{Z}$.

For $j \neq 0$, and with $\varphi(h_i) = -\gamma_i$ and $\varphi(h_k) = \varphi(h_k^+) - \gamma_k$ we conclude $|\varphi(h_k) - \varphi(h_i)| = |2k\pi - \gamma_k + \gamma_i| > \pi$, which contradicts to the $\mathcal{C}_{\text{free}}$-condition.
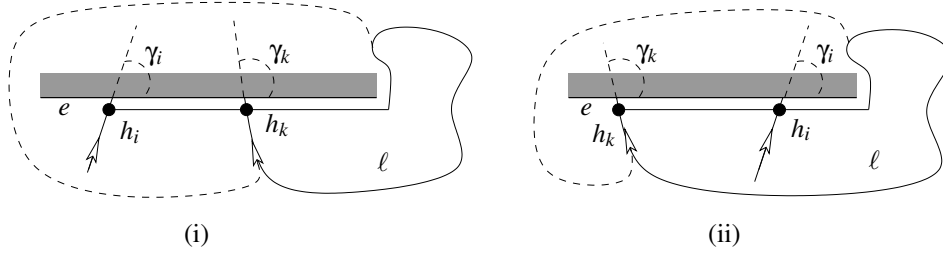
Figure 2.7: A curve from $\mathcal{K}$ hits any edge once.

Therefore we conclude $j = 0$ and also $\varphi(h_k^+) = 0$. But we can argue that there be a full cw or ccw turn from $P(h_i)$ to $P(h_k)$ without intersections; see Figure 2.7. The curve C made a full turn with angular counter change of $\pm 2\pi$. This means that $\varphi(h_k^+) = \pm 2\pi$ should hold. $\lightning$ $\qquad\square$

By Lemma 2.7 we can now prove that the condition from Definition 2.5 are sufficient. First, we require a helping lemma. If the curve gets stuck onto a single obstacle, the obstacle should enclose the curve.

Lemma 2.8 *If a curve $C \in \mathcal{K}$ does not leave an obstacle anymore the curve is enclosed by the obstacle.*

Proof. If the curve does not leave an obstacle after the last hit-point, the path along the boundary is repeated infinitely often. The path can be in cw or in ccw order which means a counter change of $+2\pi$ for ccw order or a counter change of -2π in cw order for any round. In the first case at some point the counter gets arbitrarily large for any point on the boundary. So also for the last hit-point and the C_{half}-condition is violated. This means we can only have a cw order loop and by the Left-Hand-Rule the curve has to be enclosed by the obstacle. $\qquad\square$

Theorem 2.9 *(Kamphans, Langetepe, 2003)*
An agent, who follows a path from $C \in \mathcal{K}$ will escape from any labyrinth and from any position, if an escape path exists. *[KL03]*

Proof. If there is an escape path the agent and the curve is not enclosed by an obstacle. Therefore the curve $C \in \mathcal{K}$ will leave any obstacle after a while. Since any edge is hit at most once by Lemma 2.7 there will be no hit any more after a while. The C_{free}-condition takes care that the agent steadily moves into a halfplane w.r.t. a given direction. Thus we will escape from the environment. $\qquad\square$

2.1.3 Applications

We would like to consider the impact of Theorem 2.9. Let us first assume that we make use of a compass for counting the total turns around the obstacles and for holding the direction in C_{free}. If the deviation from the starting direction is never larger than $(-\pi/2, \pi/2)$, such a compass will help us leaving the labyrinth.

In this case in C_{free} we can guarantee an angular range $(-\frac{\pi}{2}, +\frac{\pi}{2})$. Along the boundary the absolut error is smaller than $|\frac{\pi}{2}|$, the maximal positive value along the boundary is smaller than $+\frac{\pi}{2}$, the minimal value in C_{free} is larger than $+\frac{\pi}{2}$, this yields the C_{half}-condition.

$$\forall h_i, t \in C : P(t) = P(h_i) \Rightarrow \varphi(t) - \varphi(h_i) < \pi\,!$$

Corollary 2.10 *The usage of a compass with absolut deviation smaller than $\frac{\pi}{2}$ will help to leave a labyrinth by a pledge like algorithm.*

Next assume that we would like to make use of small deviations of the environment itself. Therefore, we consider obstacles that consists of axis-parallel edges, only. For such orthogonal polygons, we can

simply sum up the reflex vertices (inner angle $> \pi$, $+1$) and the convex vertices (inner angle $< \pi$, -1) as indicated and will leave an obstacle with counter value 0; see Figure 2.8(i). If we guarantee a deviation in the range $(-\frac{\pi}{2}, +\frac{\pi}{2})$ in the free space, we can successfully apply the pledge algorithm. After a hit, we only have to find out whether the edge is horizontal or vertical. For vertical edges we simply slip along the edge and wait for the next hit. Thus we start the movement along the boundary with angular counter $+1$, following a horizontal edge.
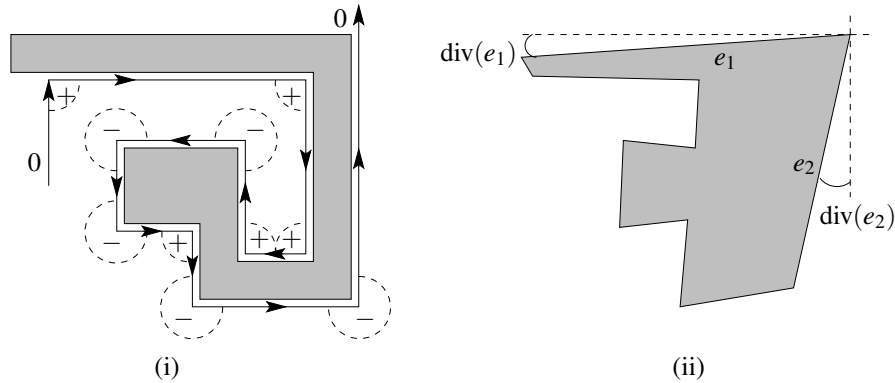

Figure 2.8: (i) Counting the angles in orthogonal polygons, (ii) pseudo-orthogonal polygon with deviation div.

More generally for such a counting argument we have to take care that: Folgende drei Bedingungen sind also einzuhalten:

1. Reflex and convex vertices can be distinguished: Count the rotation correctly.

2. Maximal deviation from the starting direction. Interval of size π.

3. Distinguish: Horizontal/verticale edge.

Now let us assume that the polygons are not exactly axis-parallel but roughly as shown in Figure 2.8(ii).

By, $\operatorname{div}(e)$, for edge Kante $e = (v, w)$ we define the smallest deviation from a vertical or horizontal edge passing through w or v. This deviation should be small in total for all edges. Additionally, we would like to take care that the number of reflex and convex vertices of the polygon fits to an axis-parallel polygon.

Definition 2.11   A polygonal scene is δ-pseudo orthogonal for $\delta > 0$, if for any polygon we have (number if convex vertices) = (Number of reflex vertices) $+ 4$ and $\operatorname{div}(P) := \max_{e \in P} \operatorname{div}(e) \leq \delta$ holds.

For δ-pseudo orthogonal we would like to proceed as indicated above and have to fulfil the above three conditions. Let us additionally assume that the angle counter device on the obstacle has a deviation (error) of no more than ρ. The following corollary shows some legal value for ρ and δ also with the interplay of the free space movements.

Corollary 2.12   *Let P be a δ-pseudo orthogonal scene and let us assume that we count the angles with precision ρ such that $\delta + \rho < \frac{\pi}{4}$. If we take care that the deviation from the starting direction in C_{free} is no more than $\frac{\pi}{4} - 2\delta - \rho$, the simple reflex/convex counter pledge like algorithm helps us to leave a labyrinth.*

Proof. We have to distinguish between reflex and convex vertices at the boundary, otherwise the $+1$, -1 counting will be erroneous. We consider the worst-case situation for our measurement. Let us assume that at a vertex we measure the outer angle γ as shown in Figure **??**. For $\gamma < \pi$ we assume that we have a reflex vertex and for $\gamma > \pi$ we assume that the vertex is convex.

We would like to guarantee a correct detection for tha maximal error and deviation. The correct angle of a convex vertex is $\frac{3}{2}\pi$ and the correct angle of a reflex vertex is $\frac{\pi}{2}$. Therefore we require:

$$\frac{3}{2}\pi - 2\delta - \rho > \pi \qquad \text{und} \qquad \frac{\pi}{2} + 2\delta + \rho < \pi$$
$$\Leftrightarrow 2\delta + \rho < \frac{\pi}{2} \qquad\qquad \Leftrightarrow 2\delta + \rho < \frac{\pi}{2}.$$

Additionally, we would like to distinguish between horizontal and vertical edges after hitting an edge. Either we slip along the vertical edge or we start at the horizontal with the simple counter. Again we assume the worst-case situation; see Figure 2.9.

We measure the turning angle γ for the corresponding edge. For a horizontal edges this is exact $-\frac{\pi}{2}$; see Figure 2.9(i). If this angle is between $-\frac{\pi}{4}$ and $-\frac{3\pi}{4}$ we conclude that we have a horizontal edge. Otherwise the edge is assumed to be vertical. Note that γ is always negative. We assume that the deviation from the starting direction is φ.
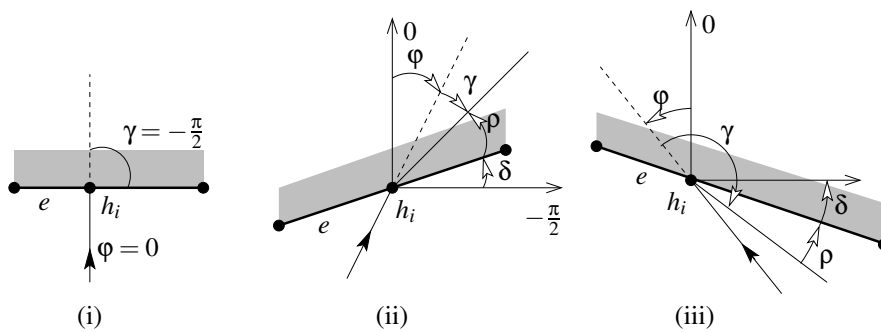

Figure 2.9: Hitting a horizonal edges (i) in the error-free case, (ii) for small absolut γ, (iii) for large absolut γ.

In Figure 2.9(ii) the deviations φ, δ and ρ should make $|\gamma|$ as small as possible and still smaller than $-\frac{\pi}{4}$, φ is negative. In Figure 2.9(iii) the deviations φ, δ and ρ should make $|\gamma|$ as large as possible and larger than $-\frac{3\pi}{4}$, φ is positive. We conclude from Figure 2.9(ii)

$$\gamma = -\frac{\pi}{2} - \varphi + \delta + \rho < -\frac{\pi}{4} \;\Leftrightarrow\; -\frac{\pi}{4} + \delta + \rho < \varphi,$$

and from Figure 2.9(iii)

$$\gamma = -\left(\frac{\pi}{2} + \varphi + \delta + \rho\right) > -\frac{3}{4}\pi \;\Leftrightarrow\; \frac{\pi}{4} - \delta - \rho > \varphi.$$

We We detect horizontal edges precisely if $\varphi(h_i) \in \,]-\frac{\pi}{4} + \delta + \rho, \frac{\pi}{4} - \delta - \rho\,[$ holds. Therefore we require $\delta + \rho < \frac{\pi}{4}$. A maximal deviation of $\frac{\pi}{4} - \delta - \rho$ would be enough for correct detections. Since we might start the free space move with an error of δ at a vertex we require $\frac{\pi}{4} - 2\delta - \rho$ for the deviation. $\square$

Exercise 15 *In the above corollary we can set $\delta = 0$ and $\rho = 0$ and require that we do not deviate in the free space by an angle of $\pi/4$. Why is this different to the error-free case where an error of less than $\pi/2$ was allowed for the free space movements.*

2.2 Navigation with touching sensor

We distinguish between the term Navigation for visiting a given target (known coordinates) and the term Searching for searching for an unknown goal (unknown coordinates). The family of the so-called Bug-Algorithms are the first algorithms for the navigation task in polygonal environments[2]. The first

---

[2]In this case Bug is not meant as a synonym for an error.

simple strategies have been introduced by Lumelsky and Stepanov [LS87], extensions and modifications came from Sankaranarayanan et al. [SM92, SV90a, SV90b, SV91]. Many variants have been discussed since then. Bug-variants have been practically used for the navigation of some of the Mars rovers like Sojourner or Bridget, (see also RoverBug, [LB99]).

In the following we assume that the coordinates of the target are known and that the agent have a finite storage so that coordinates of points and /or length of (sub)path can be stored. The agent also is aware of the coordinates of its current position, for example by GPS.

Any Bug-algorithm runs with the same principle and actions: The agent moves toward the target until an obstacle is visited (Move-To-Target action) Then the agent follows the wall of the obstacle for a while (Follow-Wall action) until some condition triggers the next movement in the free space toward the target. The leaving condition is the main difference between the Bug-variants.

We assume that the agent R is point-shaped and equipped with a touch sensor for the Follow-Wall action. We make use of the following notations:

- $|pq|$ denotes the distance between two points p and q,

- $D := |st|$ denotes distance from start s to target t,

- π_S denotes the path of a strategy S from s to t; $|\pi_S|$ denotes the length of this path where $|\pi_S| := \infty$, if there is no such path,

- $U(P_i)$ denotes the perimeter of the obstacle P_i.

2.2.1 Strategies of Lumelsky and Stepanov

The first algorithm Algorithm 2.2, Bug1, leaves an obstacle P_j at a point $p \in P_i$ that is the closest point to the target. This defines a sequence of **Hit-Points** h_i, where the agent hits an obstacle and **Leave-Points** ℓ_i, where the agent leaves an obstacle. Since the coordinates of the target and the coordinates of the current position are known, the agent can calculate the corresponding distances. Additionally, by successively counting small steps, the agent can calculate the path length of the path along the boundary during the circumnavigation and also the path length to the currently computed optional leave-point. Additionally, the path length (along the boundary) to the With these values the agent can perform step 3 of Algorithm 2.2 Figure 2.10 shows an example for the path of Bug1.
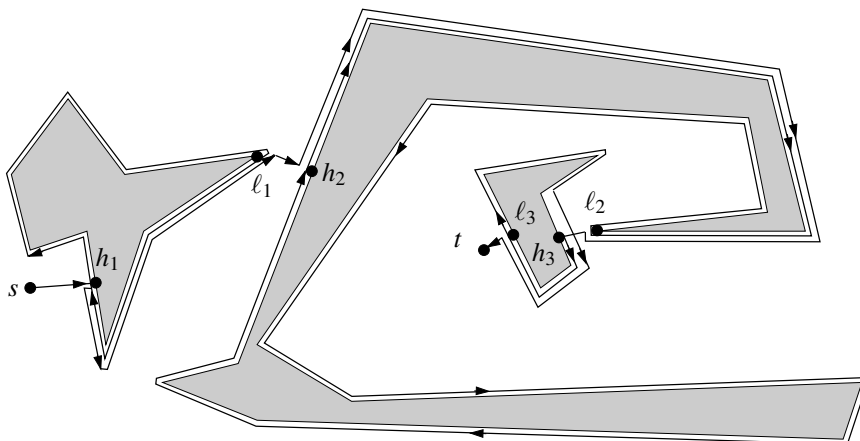

Figure 2.10: Example execution of strategy Bug1.

We assume that there is a finite number of polygonal obstacles and that the obstacles do not touch or intesect. The number of polygons is finite in the sense that any circle of fixed radius contains only finitely many *obstacles* P_i.

Algorithm 2.2 Bug1

---

0. $\ell_0 := s$, $i := 1$

1. From ℓ_{i-1} move toward the target, until

   (a) Target is visited: Stop!

   (b) An obstacle is reached at point h_i. If $h_i = \ell_{i-1}$: The goal cannot be reached.

2. Surround the obstacle in cw-order — keep track of the point ℓ_i on the boundary with the shortest distance to t —, until

   (a) Target is visited: Stop!

   (b) h_i is reached.

3. Move along the shortest path along the boundary to ℓ_i.

4. Increase i, GOTO 1.

Theorem 2.13 *(Lumelsky, Stepanov, 1985)*
Strategy Bug1 finds a path from a starting point s to a target t, if such a path exists. [LS87]

Proof. For the sequence of hit- und leave-points we have

$$|st| \geq |h_1 t| \geq |\ell_1 t| \ldots \geq |h_k t| \geq |\ell_k t|.$$

Since for any visited obstacles we choose a leave-point that is closest to the target, any obstacles can be left. Otherwise, if this is not the case, the obstacle would fully enclose the target. This also means that we have a strict $>$ in the above sequence. Any obstacle can be visited only once and the finite number of obstacles within the circle of radius $|st|$ around t lead to a finite sequence which ends at the target. $\square$

For the performance we conclude:

Theorem 2.14 *(Lumelsky, Stepanov, 1985)*
Let π_{Bug1} denote the path from s to t, for the successful application of the strategy Bug1. [LS87] *We have:*

$$|\pi_{\text{Bug1}}| \leq D + \frac{3}{2} \sum_i U(P_i).$$

Proof. We subdivide the path into the movements along the boundary of the obstacles P_i and the movements in the free space. Since step 3. of Algorithm 2.2 makes use of a shortest path we have path length $\frac{3}{2} \sum U(P_i)$ for any visited obstacle. It remains to calculate the length D' for the free space movements. We show that $D' \leq D$ holds.

$$
\begin{aligned}
D' &= |sh_1| + |\ell_1 h_2| + \ldots + |\ell_{k-1} h_k| + |\ell_k t| \\
&\leq |sh_1| + |\ell_1 h_2| + \ldots + |\ell_{k-1} h_k| + |h_k t| \\
&= |sh_1| + |\ell_1 h_2| + \ldots + |\ell_{k-1} t| \\
&\cdots \\
&\leq |sh_1| + |\ell_1 h_2| + |h_2 t| \\
&= |sh_1| + |\ell_1 t| \\
&\leq |sh_1| + |h_1 t| = |st| = D
\end{aligned}
$$

We can compare the above result with the lower bound Theorem **??** and conclude that in comparison to any other Bug-strategy the strategy Bug1 can be consodered to be $\frac{3}{2}$-competitive.

Corollary 2.15 *Bug1 is $\frac{3}{2}$-competitive in comparison to arbitrary Bug-like online strategyies.*

In the next variant we would like to avoid complete circumnavigations of the obstacles. Therefore we make use of a line G passing through the segment st. At any time during the Wall-Follow action we will try to move toward the target if we reach a point at G that is closer to t than the previous hit-point; see Algorithm 2.3. Note that by this action, it is possible to visit an obstacles more than once which was impossible for Bug1. h_j and ℓ_j do no longer denote hit- and leave-points of the j-th obstacle.
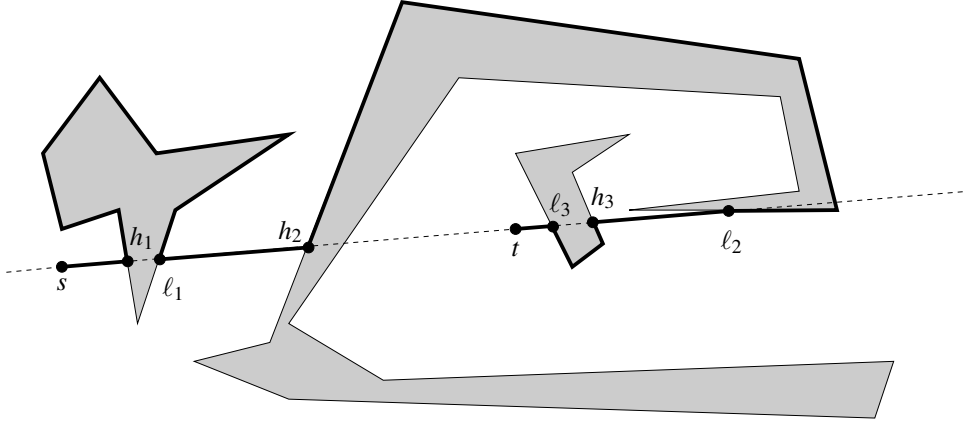
Figure 2.11: Example of the execution of the strategy Bug2.

Figure 2.12 shows an example, where an obtacle is visited more than once. After hit-point h_3 the agent does not leave the obstacle at p_1 or ℓ_1 since $|h_3 t|$ is smaller thanthe distance of p_1 and ℓ_1 to t. At p_2 and p_3 the agent cannot leave the obstacle since the segments $\overline{p_{2/3}t}$ are blocked by the obstacle.
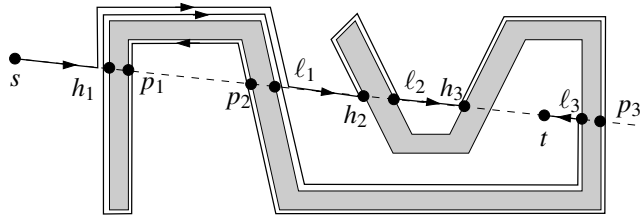

Figure 2.12: The execution of Bug2 can lead to several visits of the same obstacle.

The number of polygons is finite in the sense that any circle of fixed radius contains only finitely many *obstacles P_i.*

Lemma 2.16 *The strategy Bug2 meets finitely many obstacles.*

Proof. In step 2b of Algorithm 2.3 the agent leaves an obstacle only if $|\ell_j t| < |h_j t|$ holds. Since the circle of radius $|\overline{st}|$ around t contains only finitely many obstacles we can hit only finitely many obstacles.   □

The number of surroundings depend on the intersections of the line passing through st with the boundary of the relevant obstacles.

Lemma 2.17 *Let n_i denote the number of intersections of the line $\overleftrightarrow{st}$ passing through st with the boundary of polygon P_i. The strategy Bug2 visits any point of P_i only $\frac{n_i}{2}$ times.*

Proof.
Bug2 successively defines pairs (h_j, ℓ_j) of hit- und leave-points and by the leave condition we have

$$|h_j t| > |\ell_j t| > |h_{j+1} t|.$$

Algorithm 2.3 Bug2

---

0. $\ell_0 := s$, $j := 1$

1. From ℓ_{j-1} move toward the target, until

   (a) Target is reached: Stop!

   (b) An obstacle is visited at h_j.

2. Surround the obstacle in cw-order, until

   (a) Target is reached: Stop!

   (b) The line passing segment *st* is visited at point q, $|qt| < |h_j t|$ and $\overline{qt}$ is *free*, such that we can leave the obstacle from q toward the target.
   Set $\ell_j := q$, $j := j + 1$ and GOTO 1.

   (c) h_j is visited again without reaching a point q as in described in b). The target cannot be reached. erreicht werden.

This means that any point is only once a leave-point or a hit-point and any intersection point can appear only in one pair (h_j, ℓ_j). On the other hand a single pair can only lead to one full surrounding, if the same hit-point is visited, the strategy stops. We have at most $\frac{n_i}{2}$ pairs and surroundings. □

Finally we conclude that we have only finitely many relevant intersections and either the strategy visits a current hit-point again and the corresponding obstacle enlcoses the target or we will finally succeed.

Corollary 2.18   *Strategy Bug2 is successful, if the target can be reached.*

The performance of Bug2 is given in the following statement.

Theorem 2.19   *(Lumelsky, Stepanov, 1985)*
Let π_{Bug2} denote the path from s to t, for the successful application of strategy Bug2. We have

$$|\pi_{\text{Bug2}}| \leq D + \sum_i \frac{n_i \, \text{U}(P_i)}{2}.$$

Here P_i is an obstacle that is visited during the execution of Bug2.   *[LS87]*

Proof. The term $\sum \frac{n_i \, \text{U}(P_i)}{2}$ follows from Lemma 2.17. For the length of the free space movements, say D', between the obstacles, we make use of the same arguments as in the proof of Theorem 2.14 and conclude $D' \leq D$. □

Bug2 is not always better than Bug1. Obviuously, in the presence of convex obstacles, Bug2 outperforms Bug1.

Corollary 2.20   *For a polygonal scene with convex obstacles the successful application of strategy Bug2 has path length*

$$|\pi_{\text{Bug2}}| \leq D + \sum_i \text{U}(P_i).$$

Exercise 16   *Compare the variants Bug1 and Bug2. Present an example where Bug1 outperforms Bug2. Show that for both strategies the performance guarantee is tight.*

2.2.2   Strategies from Sankaranarayanan and Vidyasagar

Many variants of the Bug-strategies have been discussed. Many of them make use of more sensor power for local improvement. For example a VisBug2 strategy makes use of a visibility range and can find local short-cuts for the Bug2 path. We would like to mention some structural different variants from Sankaranarayanan and Vidyasagar. The reason is that we would like to show that some local optimization can have unexpected disadvatages.
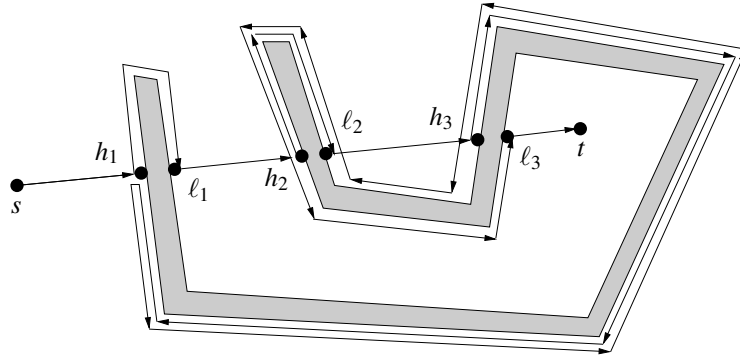

Figure 2.13: Example of the execution of Change1.

Bug1 fully surrounds any obstacle, Bug2 tries to avoid this by moving toward the goal a bit earlier. In this case a single obstacle can be visited many times. Algorithm 2.4 tries to avoid this behaviour: If a surrounding is started, and an old hit- or leave-point (not the current hit-point!) is visited, the strategy starts moving along the boundary in ccw-order; see Figure 2.13.

Theorem 2.21   *(Sankaranarayanan, Vidyasagar, 1990)*
For the length of the path of the successful application of strategy Chang1 we have   *[SV90a]*

$$|\pi_{\text{Change1}}| \leq D + 2 \cdot \sum_i \text{U}(P_i).$$

Proof. Exercise   □
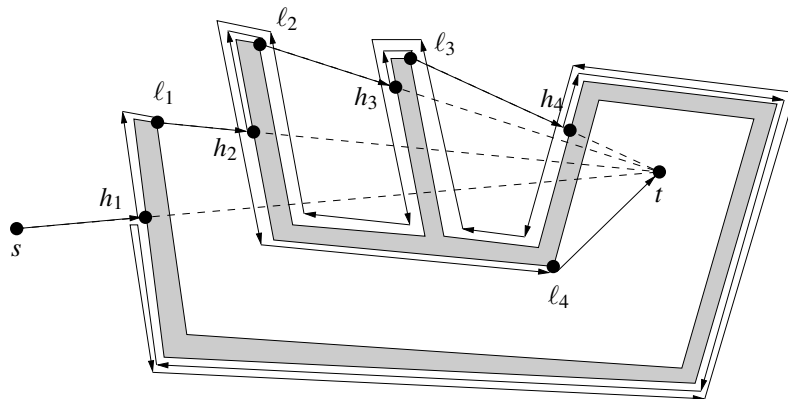

Figure 2.14: Example execution of strategy Change2.

Strategy Change2 (Algorithm 2.5) differs from Change1 only in the leaving condition. The leave-point is not restricted to a point on the line $\overleftrightarrow{st}$. As soon as there is a point q on the boundary in the Follow-Wall action that is closer to the target than the distance $|ht|$ for the last hit-point, we will leave the obstacle toward the target, if this is possible. Note that such a behaviour can also be used for a variant of Bug2.

Theorem 2.22 *(Sankaranarayanan, Vidyasagar, 1990)*
For the length of the path of the successful application of strategy Chang1 we have   *[SV90b]*

$$|\pi_{\text{Change2}}| \leq D + 2 \cdot \sum_i \text{U}(P_i).$$

Proof. Exercise   □

Exercise 17 *Present proofs for the above two Theorems. Show that the bounds are tight.*

Algorithm 2.4 Wechsel1

---

0. $\ell_0 := s$, $i := 1$

1. Move from ℓ_{i-1} along the line passing s and t toward the target, until

   (a) Target is reached: Stop!

   (b) An obstacle is reached at h_i.

2. Surround the obstacle, until

   (a) Target is reached: Stop!

   (b) The line passing s and t is visited a some point q such that the distance from q to t is smaller than $h_i t$ and the segment $\overline{qt}$ is *free* (see refalgobug2).
   Set $\ell_i := q$, $i := i + 1$ und GOTO 1.

   (c) A hit- or leave-point h_j or ℓ_j with $j < i$ is visited: Move back to h_i in ccw-order and start ccw-order surrounding under condition (a), (b) oder (d) (not (c) again!)

   (d) h_i is visited again without reaching a point as indicated in (b) or (c). The goal is enclosed by an obstacle.

Algorithm 2.5 Wechsel2

As Change1, but:

0. $\ell_0 := s$, $i := 1$

1. Move from ℓ_{i-1} along the line passing s and t toward the target, until

2.(b) A point q is visited such that the distance from q to t is smaller than $h_i t$ and the segment $\overline{qt}$ is *free* (see refalgobug2).
   Set $\ell_i := q$, $i := i + 1$ und GOTO 1.

Bibliography

[Ad80]   H. Abelson and A. A. diSessa. *Turtle Geometry*. MIT Press, Cambridge, 1980.

[AFM00]   E. M. Arkin, S. P. Fekete, and J. S. B. Mitchell. Approximation algorithms for lawn mowing and milling. *Comput. Geom. Theory Appl.*, 17:25–50, 2000.

[AKS02]   Susanne Albers, Klaus Kursawe, and Sven Schuierer. Exploring unknown environments with obstacles. *Algorithmica*, 32:123–143, 2002.

[BRS94]   Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal learning of an unknown environment. Technical Report A.I. Memo No. 1474, Massachusetts Institute of Technology, March 1994.

[DJMW91]   G. Dudek, M. Jenkin, E. Milios, and D. Wilkes. Robotic exploration as graph construction. *Transactions on Robotics and Automation*, 7:859–865, 1991.

[DKK01]   Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal constrained graph exploration. In *Proc. 12th ACM-SIAM Symp. Discr. Algo.*, pages 307–314, 2001.

[DKK06]   Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal constrained graph exploration. *ACM Trans. Algor.*, 2:380–402, 2006.

[GR03]   Yoav Gabriely and Elon Rimon. Competitive on-line coverage of grid environments by a mobile robot. *Comput. Geom. Theory Appl.*, 24:197–224, 2003.

[IKKL00a] Christian Icking, Thomas Kamphans, Rolf Klein, and Elmar Langetepe. Exploring an unknown cellular environment. In *Abstracts 16th European Workshop Comput. Geom.*, pages 140–143. Ben-Gurion University of the Negev, 2000.

[IKKL00b] Christian Icking, Thomas Kamphans, Rolf Klein, and Elmar Langetepe. Exploring an unknown cellular environment. Unpublished Manuscript, FernUniversität Hagen, 2000.

[IKKL05]   Christian Icking, Tom Kamphans, Rolf Klein, and Elmar Langetepe. Exploring simple grid polygons. In *11th Internat. Comput. Combin. Conf.*, volume 3595 of *Lecture Notes Comput. Sci.*, pages 524–533. Springer, 2005.

[IPS82]   A. Itai, C. H. Papadimitriou, and J. L. Szwarcfiter. Hamilton paths in grid graphs. *SIAM J. Comput.*, 11:676–686, 1982.

[KL03]   Tom Kamphans and Elmar Langetepe. The Pledge algorithm reconsidered under errors in sensors and motion. In *Proc. of the 1th Workshop on Approximation and Online Algorithms*, volume 2909 of *Lecture Notes Comput. Sci.*, pages 165–178. Springer, 2003.

[LB99]   Sharon Laubach and Joel Burdick. RoverBug: Long range navigation for mars rovers. In Peter Corke and James Trevelyan, editors, *Proc. 6th Int. Symp. Experimental Robotics*, volume 250 of *Lecture Notes in Control and Information Sciences*, pages 339–348. Springer, 1999.

[Lee61]   C. Y. Lee. An algorithm for path connections and its application. *IRE Trans. on Electronic Computers*, EC-10:346–365, 1961.

[LS87]   V. J. Lumelsky and A. A. Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.

[Sha52]   Claude E. Shannon. Presentation of a maze solving machine. In H. von Foerster, M. Mead, and H. L. Teuber, editors, *Cybernetics: Circular, Causal and Feedback Mechanisms in Biological and Social Systems, Transactions Eighth Conference, 1951*, pages 169–181, New York, 1952. Josiah Macy Jr. Foundation. Reprint in [Sha93].

[Sha93]   Claude E. Shannon. Presentation of a maze solving machine. In Neil J. A. Sloane and Aaron D. Wyner, editors, *Claude Shannon: Collected Papers*, volume PC-03319. IEEE Press, 1993.

[SM92]   A. Sankaranarayanan and I. Masuda. A new algorithm for robot curvefollowing amidst unknown obstacles, and a generalization of maze-searching. In *Proc. 1992 IEEE Internat. Conf. on Robotics and Automation*, pages 2487–2494, 1992.

[Sut69]   Ivan E. Sutherland. A method for solving arbitrary wall mazes by computer. *IEEE Trans. on Computers*, 18(12):1092–1097, 1969.

[SV90a]   A. Sankaranarayanan and M. Vidyasagar. A new path planning algorithm for a point object amidst unknown obstacles in a plane. In *Proc. 1990 IEEE Internat. Conf. on Robotics and Automation*, pages 1930–1936, 1990.

[SV90b]   A. Sankaranarayanan and M. Vidyasagar. Path planning for moving a point object amidst unknown obstacles in a plane: A new algorithm and a general theory for algorithm developments. In *Proceedings of 1990 IEEE Conf. on Decision and Control*, pages 1111–1119, 1990.

[SV91]   A. Sankaranarayanan and M. Vidyasagar. Path planning for moving a point object amidst unknown obstacles in a plane: The universal lower bound on the worst case path lengths and a classification of algorithms. In *Proc. 1991 IEEE Internat. Conf. on Robotics and Automation*, pages 1734–1741, 1991.

Index