Journal of Intelligent & Robotic Systems (2019) 95:443–458
https://doi.org/10.1007/s10846-018-0953-y

Optimising Robotic Pool-Cleaning with a Genetic Algorithm

V. Ramos Batista[1] · F. A. Zampirolli[1]

Received: 27 March 2018 / Accepted: 18 October 2018 / Published online: 20 November 2018

Abstract
Demand for Genetic Algorithms (GA) in research and market applications has been increasing considerably. This can be explained through big data and the necessity of interpreting them in an automatic, efficient and intelligent way. In the case of intelligent systems for unmanned vehicles there are two well-defined subsystems: autonomous and robotic navigation. Despite the present day's good development of the latter, taking the best decisions is an essential robot's attribute that can only be acquired with the former. This work presents a fully programmed GA for a robot to walk around a planar graph G with the highest efficiency. Each robot's action is one among five kinds of genes, and fourteen of them build a chromosome, namely a *sequence of actions* for the robot to walk all around G. The efficiency of a chromosome is given by the number of visited vertices and the amount of saved energy, which are both computed by a fitness function. Our GA returns near-optimal chromosomes for the robot to clean the whole reflection pool with the least energy consumption. Our Coverage Path Planning differs from others in the literature because they consider obtaining a near-optimal *sequence of vertices* for the robot to follow in that order. Moreover, for such a sequence they do not allow vertex repetitions, whereas in our developed algorithm the robot can pass more than once in a same vertex, with the objective of guaranteeing a lower energy consumption. This objective already makes our best chromosomes avoid repeating vertices, as we have observed in our experiments.

Keywords Cleaning robot · Reflection pools · Optimisation · Genetic algorithm · Coverage path planning

1 Introduction

Cleaning robots have several applications such as in housework [1], pool cleaning [2, 3] and sanitising of industrial facilities [4] (pipes, tanks, warehouses, etc.) These and many other tasks have several examples in the literature: driverless cars and tractors [5, 6], fruit harvesting robots [7] and autonomous glass cleaning machines [1], just to mention a few. In this paper we deal with a cleaning robot devoted to reflection pools, and since they typically have a constant depth one can still work with a two-dimensional approach. Here we present the implementation of a genetic algorithm, which complements a previous work [8] on the mathematics behind our strategies. Among other related works we cite Pichon et al. [9], Renaud et al. [10] and van der Meijden et al. [11] but to the best of our knowledge they all seem devoted to general pools. However, cleaning reflection pools brings many simplifications for a yet demanding task, hence it is worthwhile studying this special case. Indeed, a typical reflection pool is shallow and has a constant depth. If we project a 1m tall robot then its top can carry non-waterproof equipment, which is inexpensive. Moreover, our problem becomes 2D instead of 3D, and Fig. 4 in Section 5 shows how the pool is mapped by the robot itself.

Projecting a robot with autonomous navigation imposes many challenges. One of the most intricate is the *map-based navigation*. This navigation modulus involves planning near-optimal routes for the robot to accomplish its task even in the presence of obstacles. In the case of a cleaning robot the whole area previously specified as being of interest must be covered by the modulus. This one is sometimes replaced with *reactive navigation*, as in the Roomba robot's case [12]. It is a vacuum cleaner that performs a random move around a flat floor, hence without a mapped route but reacting to any obstacle by either changing direction or getting round it. After a long time one expects that the robot will have covered the whole area of interest.

✉ V. Ramos Batista
valerio.batista@ufabc.edu.br

F. A. Zampirolli
fzampirolli@ufabc.edu.br

1 Federal University of ABC, Av. dos Estados 5001, 09210-580 St André SP, Brazil

The map-based navigation may include sensors and computer vision in order to improve the robot's performance. For instance, by means of cameras and image processing a pool cleaning robot can check and decide whether a given spot needs scrubbing. This helps save energy. However, projecting and implementing this feature is highly non-trivial due to computer methods. They still can neither distinguish nuances nor make evaluations as efficiently as a human cleaner does.

Our present work is inspired in the work of Mitchell [13, Ch.9]. There the author presents a practical example of Genetic Algorithms: Robby, the soda-can-collecting robot. It is in a 10×10 array of cells, half of them empty and the rest with only one soda-can per cell. The cans are uniformly distributed. At each position Robby can check only five cells for soda-cans: North, South, West, East and Current Site. It does not know the environment and must pick all cans with only 200 drill commands, among *one cell left/right/forward/backward*, *rest* and *pick*.

Robby tries to maximise scores through a Reward-Punishment Learning Strategy: bumping into the wall gives -5 and picking a can gives $+10$, but trying it in an empty cell gives -1. Namely, it can score 500 at most. By letting Robby evolve in the Genetic Algorithm learning strategy, after 10 thousand tries it scored 483 in average. We shall resume Robby's example in Section 4.

This paper is organised as follows: in Section 2 we briefly explain our technique, which is compared to others in Section 3. In Section 4 we summarise the resources and overall strategies of our method. It consists of three main steps that are explained in Section 5. There we give some details but focus on Section 5.4, where the GA is applied to a practical example and explained step-by-step. Section 6 presents three experiments with the GA and we finally draw our conclusions in Section 7.

2 Technical Overview

In our present work we deal with map-based navigation oriented by predefined routes. However, our approach does not include computer vision yet, which will be left for future developments. For any pool contour with obstacles we obtain a corresponding map of its bottom given by an almost grid graph $G = (V, E)$. Here V and E stand for the sets of vertices and edges, respectively. It is almost a grid in the sense that most of the incident edges make an angle quite close to either $90°$ or $180°$. Obtaining such G was the main achievement of our previous work [8]. As explained there, it is much easier to program the robot's displacements within a graph with these simplifications. By the way, throughout this work we only consider *connected* graphs.

In previous work [8] we also mentioned that our robot does not deal with heavy cleaning yet. This is because of the simplifications of our present approach. In future, computer vision will help decide whether a spot needs more brushing or even none, whereas the amount of battery will also depend on how long the reflection pool has been left untidy. As one can see, heavy cleaning turns the problem much more complicated, and therefore we shall only study it in forthcoming works.

Our robot scrubs each $v \in V$ but only once, namely at the first time the vertex is accessed. The chosen $e \in E$ determine a route for the robot to pass all through the set V. The robot works with two liftable soft brushes that spread well beyond the limits of its base. This base is a 0.5×1m rectangle. Adjacent vertices in V are assured to be 35-45cm apart. Hence, each of the robot's displacement is just occasionally preceded by $\pm90°$ rotations before going straight ahead.

It is much simpler to work with a fixed rotation angle. In this case the fine-tuning of the robot's position can be just periodic. Namely, our robot does not turn by precise arbitrary angles steadily. Turning by a precise angle happens only when its move deviates from G within a tolerance margin.

3 Related Works

Among other approaches found in the literature we cite the works of Yang and Luo [14], Luo and Yang [15] and Nedjati et al. [16] and choose the works of Erson and Hu [17] and Giardini and Kalmár-Nagy [18] for specific comparisons. Similarly to ours their navigation modulus consists of planning the route in two main steps: to adopt an a priori map with a route as obtained here, and to make adjustments during execution. The second step works as follows: by comparing the map with the actual topography, whenever a relevant discrepancy is detected the modulus must correct the route. In our case this second step is part of a forthcoming work that will include tests with a toy robot in a maquette.

In Erson and Hu [17] the authors present a solution based on the *network simplex method* in order to find near-optimal paths by solving the Minimum Cost Flow (MCF) problem. They do not resort to any Genetic Algorithm (GA), as done in our case and in Giardini and Kalmár-Nagy [18]. There the authors developed a planner for autonomous systems that they called *subtour problem*, a variant of the Travelling Salesman Problem (TSP): by subdividing the whole area of interest into n parts they consider k robots, $1 \le k \le n$, so that each one will follow a local solution of the TSP for the cleaning procedure.

As we have already mentioned, our robot rotates only by right angles. Fine adjustments to G of both position and direction occur just when a tolerance margin is exceeded. This makes our work differ from all the others because their routes can adopt any angles. Another important specialty is that our robot does not endeavour to visit each vertex only once. Passing at each vertex just once comes as a natural consequence of our GA strategy: to clean the whole pool with the least possible amount of energy.

Indeed, according to our experiments the best chromosomes visit all vertices once, and just a few of them are visited twice or even three times. Hence, instead of imposing a single visit to each vertex we naturally arrived close to that through our GA strategy.

At this point we must fix a terminology for the reader not to confuse with others in the literature, since they are not unanimous. Ours is presented in Table 1. There the terms "cannot" and "may repeat" both mean "besides start and end vertices" for closed routes.

According to Table 1 the near-optimal chromosomes obtained by our GA will make the robot perform a *walk*. Its start vertex must be on a corner of the pool, as explained in previous work [8], but the end vertex does not have to. See Section 5 for details.

While the work of Giardini and Kalmár-Nagy [18] is devoted to obtaining near-optimal cycles for the k robots, in Erson and Hu [17] the authors seek for near-optimal paths. As in our case, their robot does not finish cleaning at the start point either.

4 Materials

For the time being, our main resources are the works of Mitchell [13] and Driscoll and Trefethen [19], the softwares *Matlab*, *Surface Evolver* and the programming language C/C++11. Now we briefly describe each of them, and the reasons that lead us to combine these resources within an overall strategy to find the near-optimal cleaning procedure.

At the Introduction we mentioned the work of Mitchell [13, Ch.9], where the author introduces Robby, the soda-can-collecting robot. Section 5 explains some major

Table 1 Types of Route

Name	Vertices	Edges	Open/Closed
Circuit	may repeat	cannot	Closed
Cycle	cannot	cannot	Closed
Path	cannot	cannot	Open
Trail	may repeat	cannot	Open
Walk	may repeat	may repeat	Any

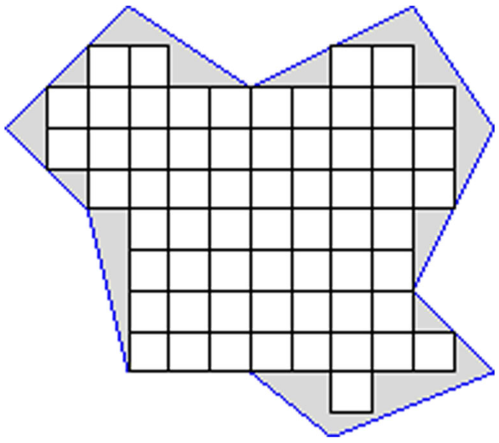

Fig. 1 A Euclidean (standard) quadrangulation; from [8, Fig.3]

adjustments we made in Robby's example to obtain our Genetic Algorithm (GA). For now we remark that reflection pools can have pretty irregular shapes and include obstacles. Hence a Euclidean quadrangulation causes many inconveniences when we strive for optimising the cleaning procedure. See Fig. 1 and [8] for details. That is why we resort to non-Euclidean quadrangulations, which can be obtained via [19]. See Fig. 2.

In general non-Euclidean quadrangulations are quite irregular, as depicted in Fig. 2b and c. In order to get an almost grid graph G we need to equalise the quadrangulation. This procedure is shown in Fig. 3a-c, which were obtained through the *Surface Evolver*. This is a general-purpose simulator for implementing innumerous physical experiments [20]. The *Surface Evolver* works with command lines and scripts written in a built-in language with a very accessible syntax. Since its first release in 1989 it has developed to a programming language. Nowadays the scripts are called *Evolver-programs*. In Section 5 we give a brief description of the equalisation procedure.

For the time being our robot is called *Scrubbrushy*. Its rectangular base is 0.5×1 meter and it has two brushes that rotate oppositely. They sweep dirty water into a suction pipe, then into a dirt container where the water passes through a filter (stiff gauze), and gets back clean to the pool. Scrubbrushy gets a graph $G = (V, E)$ from a quadrangulation as shown in Fig. 3c. Here V corresponds to Robby's cells but with as few vertices as possible.

The *Surface Evolver* gives G as a list of vertex and edge structures. Each edge structure has two pointers to vertex structures. Namely, each edge e points to its extremes v_1, v_2, which are *neighbouring vertices*. Moreover, G has the property that any pair of neighbouring $v_1, v_2 \in V$ has a Euclidean distance $d(v_1, v_2) \in [35, 45]$cm. See Fig. 3c.

As mentioned in Section 2 the robot works with two liftable soft brushes that spread well beyond its base. This

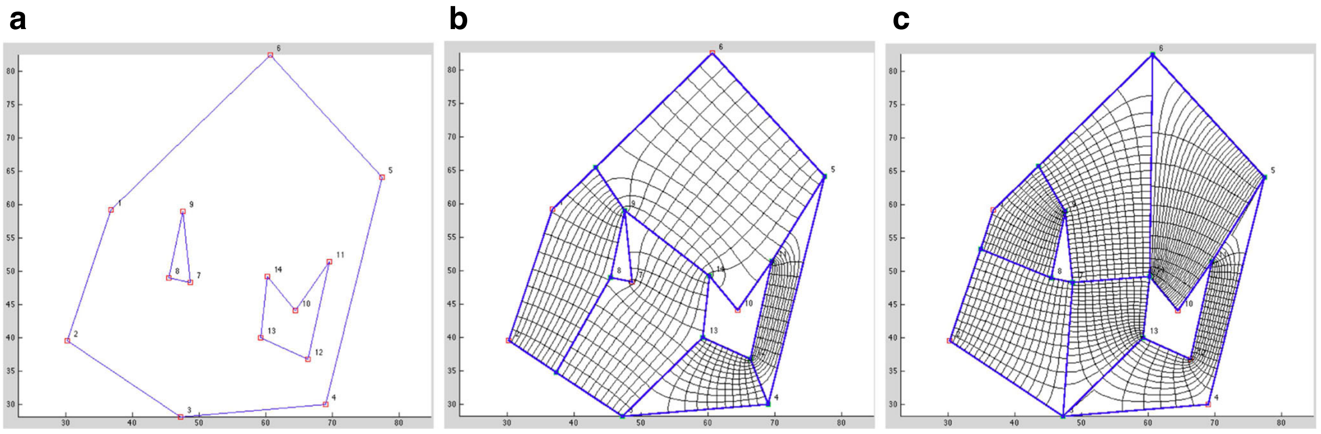

Fig. 2 Pool with obstacles (left), non-Euclidean quadrangulations (centre, right); from [8, Fig.5]

guarantees that no trace of dirt will remain inside any cell. Differently from Robby, Scrubbrushy starts with a credit that corresponds to a full charged battery in Joules. Of course, our first tests will be with a toy robot in a maquette. Then the actions and scores are: cleaning a dirty spot gives -4J, going forwards, sideways or backwards give -1J, -2J and -3J, respectively.

It is known that a 1kg body accelerates 1m/s^2 with 5J in 1s, with 10J in 2s, and so on. Hence our scores are compatible with a toy, although the exact values will only be adjusted in future. Scrubbrushy only stops when either all spots have been cleaned or the battery is empty. With our GA we find near-optimal chromosomes that make Scrubbrushy stop before the second case happens. We have chosen to programme the GA in C/C++11 in order to speed up the selection of the best chromosomes, together with some graphic visualisation.

5 Methods

As mentioned in Section 3, our robot does not endeavour to visit each vertex only once. This is because we have implemented a GA with the following setup: Each chromosome consists of a sequence of *actions* (genes) for the robot to pass through the vertices $v \in V \subset G$, and the arrangement of these actions represents the chromosome's efficiency. This one is computed by a double fitness function `f`. For each chromosome `f` gives an ordered pair where the 1st and 2nd coordinates indicate how many vertices were cleaned and how much energy was saved, respectively. Our GA returns near-optimal chromosomes for the robot to clean the whole reflection pool with the least energy consumption.

We also commented in Section 3 that the robot's start vertex must be on a corner of the pool, but the end vertex does not have to. This is because the robot gets $G = (V, E)$ implemented for each $(v, e) \in G$ to be a pair of *structures* in our GA. These include a member that indicates whether the element belongs to the pool contour. Once the whole pool has been cleaned the robot can go straight to a contour vertex up to getting round some obstacles. Figure 4a-b show how the robot identifies these elements when it is going to clean a pool for the first time (see details in [8]).

In this section we explain the three main parts of our method. The first part consists of the GA preprocessing in Sections 5.1 and 5.2, together with its logistics in Section 5.3. The second part is the proper GA, which we

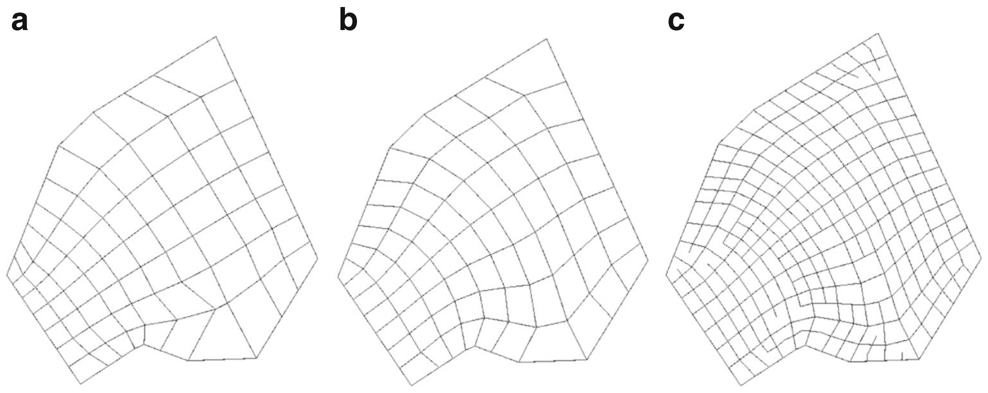

Fig. 3 Equalising and refining a non-Euclidean quadrangulation

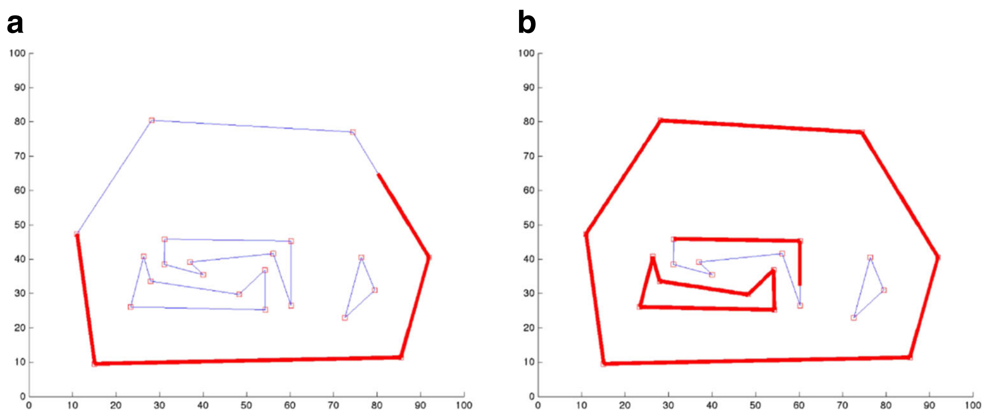

Fig. 4 Identification of main contour (left) and obstacles (right); from [8, Figs.1-2]

illustrate with a practical example in Section 5.4. The third part begins in Section 5.5, where we explain why the GA is devoted to sequences of actions rather than of vertices. Finally the structure of the code, and how to run it, are detailed in Sections 5.6 and 5.7, respectively.

5.1 Non-Euclidean Quadrangulation

As explained in previous work [8] firstly Scrubbrushy maps the contours of the reflection pool and of the obstacles therein. The contour components are then processed by the *Schwarz-Christoffel Matlab Toolbox*. See Driscoll and Trefethen [19] for details. In previous work [8] we had already obtained a user-friendly graphical interaction in Matlab called `quadr`, devoted to drawing and saving pool contours, including the user's task of choosing strategic points to quadrangulate the pool. This task is for now the only part of `quadr` that remains semi-automatic. See Fig. 2a-c.

That semi-automatic part is necessary because the Schwarz-Christoffel Toolbox cannot handle multiply connected regions. They arise when the pool has more than one contour component. In future we shall make it automatic as part of our GA.

The next subsections present the new features that have been included in our program since the previous work [8].

5.2 Equalisation

In Section 4 we explained that Scrubbrushy gets a graph $G = (V, E)$ to move around the whole bottom of the pool. However, G must have the property that any pair of neighbouring $v_1, v_2 \in V$ has a Euclidean distance $d(v_1, v_2) \in [35, 45]$cm, quite close to the robot's base dimension 0.5×1m.

Figure 2b and c show two quadrangulations obtained with `quadr` which do not have the sought after property of G. Firstly because either of them has multiple parts

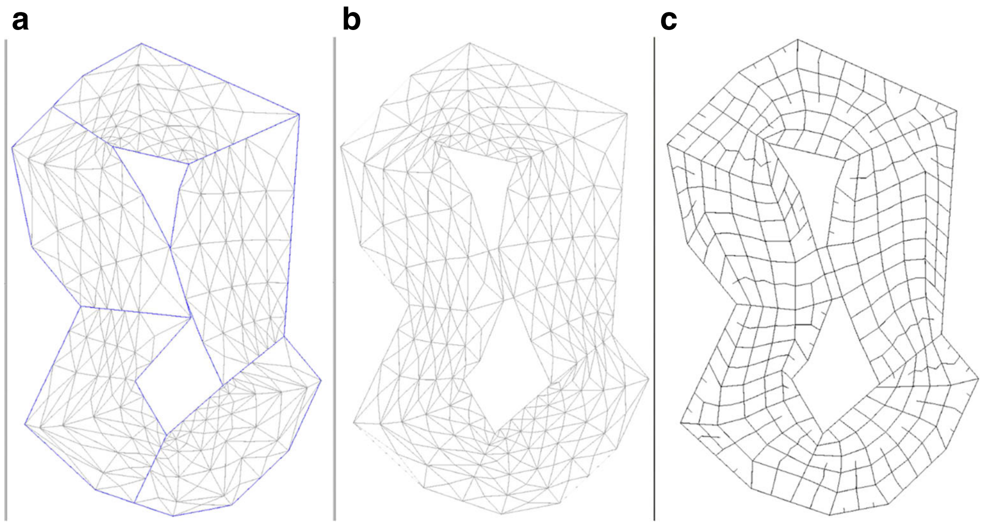

Fig. 5 Subgrids selected (left), then fitted along borders (centre) and equalised (right)

that do not fit when they meet along the blue segments. Secondly because the Schwarz-Christoffel mapping keeps right angles but can change lengths drastically.

Hence, after running `quadr` the user must invoke an Evolver-program called `equalis`. This one takes a subgrid out of each component of a quadrangulation, fits the subgrids along their junctions (represented in blue), and then alternates between refining and adjusting cell lengths to finally get neighbouring vertices apart by 40 ± 5 cm. Figure 5a-c depict these three steps with an example.

The last step that resulted in Fig. 5c is detailed in Fig. 3a-c, where `equalis` evolved another example that consisted of a single component.

5.3 Path Optimisation

Similarly to Robby, Scrubbrushy will be placed on a corner of the reflection pool and then walk all around its bottom by following a sequence S of vertices in $V \subset G = (V, E)$. The set V can have a large number $|V| = \mathtt{nv}$ of vertices, but surely E has $\mathtt{ne}$ edges such that $\mathtt{ne} < 2\,\mathtt{nv}$. This is due to the Euler-Poincaré formula and the fact that, in our case, $\mathtt{ne}$ is greater than twice the number of cells. Indeed, in the Appendix we show all the possible types of cells of our quadrangulations. Any of them has at least 4 edges, hence $\mathtt{ne} > 2\,\mathtt{nc}$, where $\mathtt{nc}$ is the total number of cells. The Euler-Poincaré formula is

$$2\,\mathtt{nv} - 2\,\mathtt{ne} + 2\,\mathtt{nc} = 4,$$

and therefore $2\,\mathtt{nv} - 4 > \mathtt{ne}$, which we simplify to $\mathtt{ne} < 2\,\mathtt{nv}$.

For a perfect cleaning S must comprise the whole V, but Scrubbrushy has to optimise energy consumption. This is because it will use a battery in order to deal even with large pools containing obstacles, whence one cannot attach the robot to the mains. But S will not necessarily consist of approximately $\mathtt{nv}$ elements, and we even expect $|S| \cong 1.25\,\mathtt{nv}$. This is because $|S|$ close to $\mathtt{nv}$ might cause the robot to spend too much battery with manœuvres.

Differently from Robby, whenever Scrubbrushy arrives at a vertex the neighbouring ones cannot be named as the cardinal points. Many of its displacements will not even be in intercardinal directions. Hence we call the next vertex by its position relative to the robot as if Scrubbrushy were a person with stretched arms: *Backmost*, *Leftmost*, *Frontmost* and *Rightmost*. In our GA code going to one of these positions are actions identified with the integers 0, 1, 2 and 3, respectively.

Scrubbrushy takes an action whenever it gets to a vertex v, but not similarly to Robby. This one does not always catch the soda-can but in our case each v accessed for the first time must be cleaned around. Otherwise the robot will take some dirt while moving to the next vertex. For the time being we assume that all vertices are dirty at the beginning, but in future Scrubbrushy will use image analysis to decide it. As mentioned in Section 2, we also consider that each v must be cleaned just once because the robot will not deal with heavy cleaning.

By looking at either Figs. 3c or 5c the reader will find vertices with different degrees, namely $1 \leq deg(v) \leq 4\ \forall v \in V$. While cleaning the pool Scrubbrushy always takes another direction after visiting v, but the number of choices are $d = deg(v)$. In the case $d = 1$ we say that v is a *spike*. Hence, if v is a spike the robot will retreat to the previous vertex. For that Scrubbrushy will turn 180° and go forward by an edge length. Namely, we have not projected it with a reverse gear.

If $d > 1$ then the robot has $d + 1$ choices: either to take one of the d directions or to cast a virtual dice. This dice corresponds to Robby's random move, a useful choice when it does not matter which direction to take (e.g. if around v the neighbouring vertices are all clean or dirty). In our case the virtual dice has d faces numbered from 0 to $d - 1$. As we have just explained, the four directions are given by 0, 1, 2 and 3. We set as 4 the action of casting a dice. See Fig. 6.

Similarly to Robby, among the neighbouring vertices Scrubbrushy can see which are clean and which are dirty. In our GA code these two cases are $\mathcal{C}$ and $\mathcal{D}$, respectively. For $deg(v) < 4$ there is a third case given by $\mathcal{N}$, namely No Thoroughfare (NT), and this case happens to as many as $4 - deg(v)$ directions.

Before going ahead we need to discuss an important detail. A zoomed crossing of Fig. 5c is shown in Fig 6, where we added labels that represent a possible way the robot will access it. Notice that coming from the backmost to the frontmost direction will force the robot to make an angle supplementary to $\theta \cong 135°$, so that it will spend a bit more than just 1J. As a matter of fact we could say that

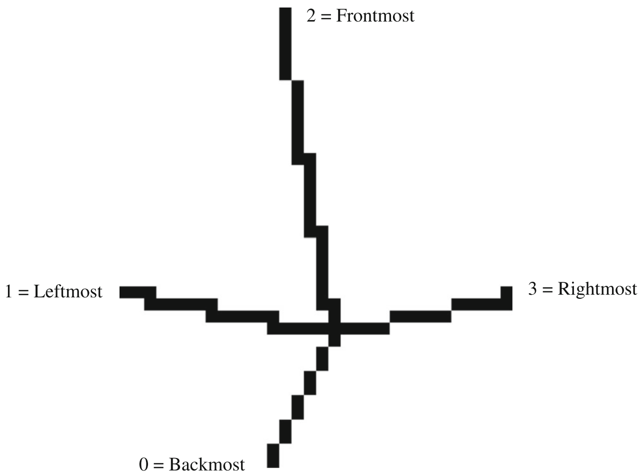

Fig. 6 Zoom of a crossing in Fig. 5c, and one possible way for the robot to access it

Table 2 The GA strategy

Array	Action	Situation ($\mathcal{D}$=dirty, $\mathcal{C}$=clean, $\mathcal{N}$=NT)		
Entry	incl. 0, 4	Leftmost	Frontmost	Rightmost
26	1, 2, **3**	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$
25	1, 2, **3**	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{D}$
24	2, **3**	$\mathcal{N}$	$\mathcal{D}$	$\mathcal{D}$
23	1, 2, **3**	$\mathcal{D}$	$\mathcal{C}$	$\mathcal{D}$
22	1, 2, **3**	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{D}$
21	2, **3**	$\mathcal{N}$	$\mathcal{C}$	$\mathcal{D}$
20	1, **3**	$\mathcal{D}$	$\mathcal{N}$	$\mathcal{D}$
19	1, **3**	$\mathcal{C}$	$\mathcal{N}$	$\mathcal{D}$
18	**3**	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{D}$
17	1, **2**, 3	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{C}$
16	1, **2**, 3	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{C}$
15	**2**, 3	$\mathcal{N}$	$\mathcal{D}$	$\mathcal{C}$
14	**1**, 2, 3	$\mathcal{D}$	$\mathcal{C}$	$\mathcal{C}$
13	1, 2, 3	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{C}$
12	2, 3	$\mathcal{N}$	$\mathcal{C}$	$\mathcal{C}$
11	**1**, 3	$\mathcal{D}$	$\mathcal{N}$	$\mathcal{C}$
10	1, 3	$\mathcal{C}$	$\mathcal{N}$	$\mathcal{C}$
9	3	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{C}$
8	1, **2**	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{N}$
7	1, **2**	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{N}$
6	**2**	$\mathcal{N}$	$\mathcal{D}$	$\mathcal{N}$
5	**1**, 2	$\mathcal{D}$	$\mathcal{C}$	$\mathcal{N}$
4	1, 2	$\mathcal{C}$	$\mathcal{C}$	$\mathcal{N}$
3	2	$\mathcal{N}$	$\mathcal{C}$	$\mathcal{N}$
2	**1**	$\mathcal{D}$	$\mathcal{N}$	$\mathcal{N}$
1	1	$\mathcal{C}$	$\mathcal{N}$	$\mathcal{N}$
0	$\star$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$

the robot spends $(2 + \cos\theta)$J, and then replace this formula with the standard values we fixed beforehand. But since we are working with approximations our GA rounds $2 + \cos\theta$ to the nearest integer.

With this formula we can shorten our chromosomes by adopting the convention that any $d > 1$ will label directions in the order Leftmost (L), Frontmost (F) and Rightmost (R). For instance, a degree two has only the Leftmost direction besides Backmost (B), with the subtlety that θ can assume any value in $]0, \pi[$. Hence, Leftmost with $\theta \cong 180°$ means "straight ahead" in practice.

Table 2 summarises all possible actions for each situation. Right after the first move the robot's backmost vertex will always be clean. For this reason the column "Backmost" was omitted from the table. Moreover, since on each line actions 0 and 4 are always possible they were considered implicitly. An exception is made for the situation $\mathcal{N}\mathcal{N}\mathcal{N}$, in which only 0 is possible. This case is indicated by $\star$. According to our convention the cells marked with light grey can be omitted from our chromosomes, so that its initial effective length 26 drops to only 14. In Table 2 we also marked some actions in bold and considered **0** where none is marked. They build a special chromosome,

chr := 333333333222100100222100100 (1)

that we shall discuss later. In our short notation we skip the 0th array entry and (1) becomes

chr = 33-33—-22-10—-22-10-10 (2)

Now we include two important exceptions $\mathcal{E}_1$ and $\mathcal{E}_2$ in Table 2.

$\mathcal{E}_1$: Whenever the robot arrives at $v \in V$ for the first time, it will check for spikes connected to v. Spikes are then cleaned prior to any other action.

$\mathcal{E}_2$: The robot will track its own trajectory, which will be *reversed* from any point where action 0 occurs, and the robot will track the reversed trajectory until encountering a situation that includes $\mathcal{D}$.

Because of $\mathcal{E}_2$ the action 0 means *more* than just a step backwards. For instance, with **chr** the robot will take that action as long as a situation does *not* include $\mathcal{D}$. This fact will be used in the Appendix to prove that **chr** makes the robot visit all vertices.

We include $\mathcal{E}_1$ for the sake of efficiency, and $\mathcal{E}_2$ as a choice of not letting the robot spend time when surrounded by clean cells. Of course, even if some few dirty spots are close to the robot, the exception $\mathcal{E}_2$ will send it away. Hopefully the robot will access these dirty spots later. Of course, $\mathcal{E}_2$ may cause some inefficiency but without it the robot could spend all the rest of its battery wandering around a large region of clean cells.

5.4 The Genetic Algorithm

Now we explain the GA with a very simple grid and a robot's trajectory generated by a chromosome named chr5,

Table 3 The chromosome chr5 in short notation

Entry	26	25	23	22	17	16	14	13	8	7	5	4	2	1
Action	0	1	4	3	2	0	1	0	4	4	1	0	1	4
Meaning	B	L	C	R	F	B	L	B	C	C	L	B	L	C

described in Table 3. We recall that 4 is the action of Casting (C) a dice.

Figure 7a shows the initial configuration with vertex id-numbers in circles and edge id-numbers in rectangles. For this grid we have $\mathtt{nv} = 14$ and we shall let the robot take at most 21 movements, namely $3\,\mathtt{nv}/2$. The general factor 3/2 is computed later in Eq. 3 but here we can already give a brief explanation: For larger grids the robot might abandon many vertices even with an unlimited number of movements. This is because of Genes 0 and 4, which can make it start a closed coming-and-going cycle.

The robot always starts by cleaning Vertex 1, and then edges around it take the clockwise order depicted in Fig. 6. Hence with respect to the robot Edges 1 and 2 are behind and on the left, respectively. This is Situation $\mathcal{DNN}$ in Table 2, which corresponds to the 2nd entry of a chromosome with 0, 1 or 4 as a possible action. In our example we use Table 3, hence it is Action 1 and the robot moves to Vertex 2, as shown in Fig. 7b.

Right after cleaning Vertex 2 the robot encounters Situation $\mathcal{DDN}$, and now the 8th entry of chr5 in Table 3 is 4, namely "Cast a dice". In our example the dice showed 0, hence the robot turns 180° and goes back to Vertex 1 with Situation $\mathcal{DNN}$ again. But this time it is Vertex 6 on the "leftmost" direction. Indeed, as explained in Section 5.3 the missing directions 2 and 3 of Fig. 6 leave *denominations* only for 0 and 1. Hence, our convention is now to call Vertex 6 as the "leftmost" one. The robot goes there, cleans it and again we have Situation $\mathcal{DNN}$, as indicated in Fig. 7c. This repetition makes the robot move to Vertices 4 and then 7, where we have $\mathcal{DDN}$ again (see Fig. 7d).

But this time the dice showed 2. Now there is not "rightmost", whereas "frontmost" and "leftmost" are Vertices 13 and 5, respectively. The next step is then Vertex 13, which has Vertex 10 as a spike. Because of $\mathcal{E}_1$ the robot goes there to clean it (see Fig. 7e). Then it rotates 180°, returns to Vertex 13 and takes the *same* position it had before having found the spike. Namely, now the robot encounters Situation $\mathcal{DCC}$ and since the 14th entry of chr5 is 1 then it goes to Vertex 11 (see Fig. 7f). There it meets Situation $\mathcal{DNN}$, and by the 2nd entry of chr5 the robot goes further "leftwards" (in fact straight ahead since Vertex 11 has degree 2). Now at Vertex 14 we have the spike Vertex 12, which is then cleaned prior to any other neighbour. See Fig. 7g.

Back to Vertex 14, at the same position it had arrived there we find $\mathcal{DCD}$. The 23rd entry of chr5 makes the robot cast a dice, which this time shows 1. The robot goes to Vertex 8 where the Situation is now $\mathcal{CDN}$, so the 7th entry makes it cast the dice again. It shows 2 and the robot goes to Vertex 3 (see Fig. 7h).

At Vertex 3 we have $\mathcal{DNN}$ again, so the robot casts the dice and gets 0. Because of $\mathcal{E}_2$ it rewinds the trajectory back to Vertex 14, the first one with a dirty neighbour in the reversing process. We still get the same Fig. 7h but now the Situation is $\mathcal{CDC}$, namely entry 16. Hence $\mathcal{E}_2$ applies again until we are once more at Vertex 3 with $\mathcal{DNN}$. This time the dice showed 1, so the robot goes to Vertex 9 with $\mathcal{DNN}$ again. Casting the dice once more gave 1, and so the robot finally stops right after cleaning Vertex 5 (see Fig. 7i). In this example we ended up with a total of 19 movements.

Of course, Gene 0 is necessary because some v are spikes, and we have already discussed the importance of Gene 4 in Section 5.3. However, this gene can make the robot go round in circles. Hence, based on the wall follower strategy we have introduced the special chromosome **chr** in Eq. 1. Figure 8a-b show its output for the grids in Figs. 3c and 5c, respectively. In the Appendix we give a theoretical proof that **chr** assures the robot to visit all vertices. But there we also show that this chromosome can require a number of steps $|S|$ greater than $2.27\,\mathtt{nv}$ in some special cases.

On the one hand, the GA can terminate with selected chromosomes containing Gene 4. Such chromosomes make the robot's trajectory non-deterministic, and it can even go

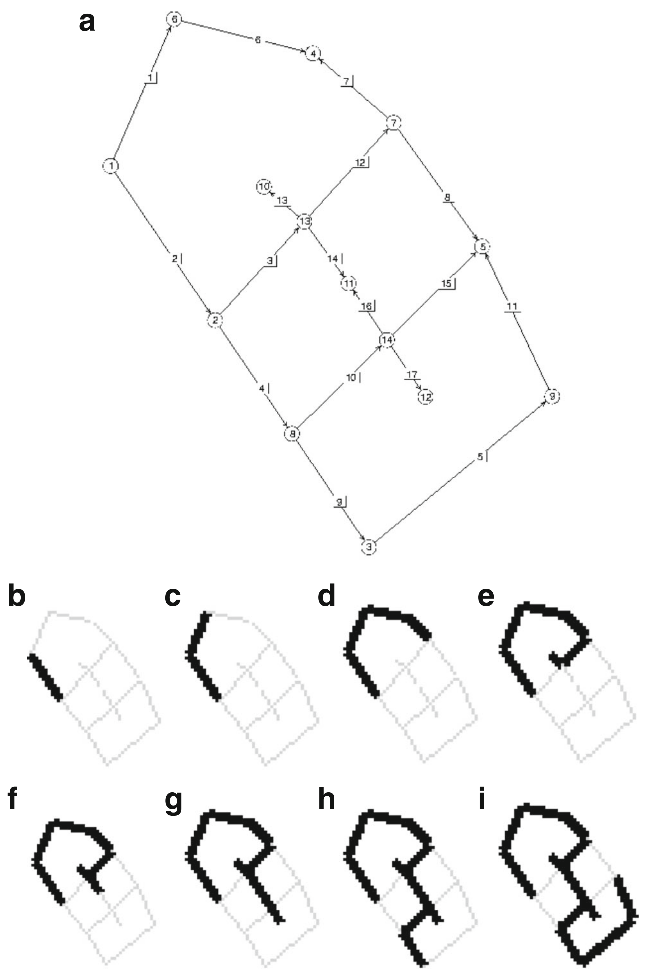

Fig. 7 Initial configuration (**a**) and some further steps (**b-i**)

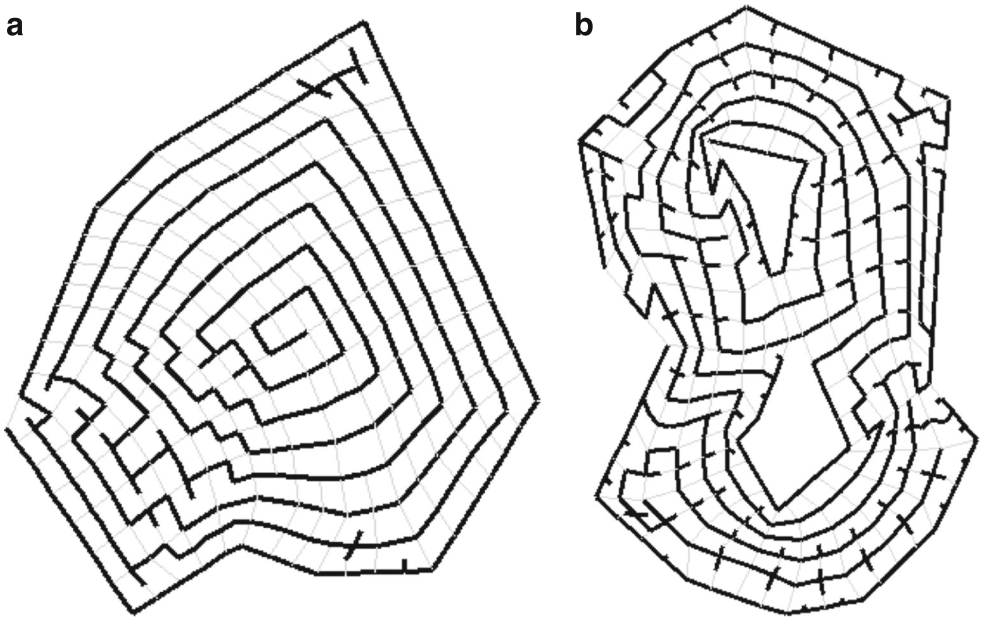

Fig. 8 Graphical outputs of **chr** for Fig. 3c (left) and Fig. 5c (right)

round in circles as we have just pointed out. On the other hand, **chr** can be too expensive and the GA will discard it rightly so. In order to resolve this deadlock the GA has to include some trajectories produced by chromosomes in each generation, so that eventually we shall have near-optimal trajectories.

An optimal trajectory makes the robot visit *all* vertices with a number of steps lesser than $1.25\,\mathtt{nv}$, as proved in the Appendix. The optimal trajectory can consume much battery, but of course we shall not be saving power if $|S|$ approaches $2.27\,\mathtt{nv}$. Hence we adopt a factor between 1.25 and 2.27 to set an upper bound for $|S|/\mathtt{nv}$ in our GA. Let us then allow the robot to give some more steps than the guaranteed $1.25\,\mathtt{nv}$, but with a certain good retreat from $2.27\,\mathtt{nv}$. For instance, weight 3 to the first and 1 to the second. Therefore

$$\frac{3 \cdot 1.25 + 1 \cdot 2.27}{3+1} \cong 1.5 \tag{3}$$

is adopted for us to take $1.5\,\mathtt{nv}$ as the upper bound of $|S|$ in our GA.

5.5 Sequence of Actions versus Sequence of Vertices

In our problem there exist as many as $N = 5^8 \cdot 4^{12} \cdot 3^6 \cong 4.8 \cdot 10^{15}$ chromosomes, or $N = 5^8 \cdot 4^4 \cdot 3^2 = 9 \cdot 10^8$ in the short notation. On the one hand, if our chromosomes were a sequence of vertices then Genes 0 and 4 would not be necessary. On the other hand, for large $\mathtt{nv}$ they would be an $\mathcal{O}(3^{\mathtt{nv}})$ function, hence of much higher complexity than our approach.

As already explained in Section 4 our work is inspired in the can-collecting robot Robby from Mitchell [13, Ch.9]. There the author's best chromosomes also include backward and random moves, which in our case correspond to precisely Genes 0 and 4. Hence our best chromosomes are given by a sequence of *actions* as a first step. As a second step these ones give near-optimal trajectories. Each of them is a sequence of *vertices* and its length approaches $1.25\,\mathtt{nv}$, much longer than the fixed chromosome size 14.

Of course, it is impossible to check the numerical performance of all chromosomes, including the many trajectories that each one can produce at casting the dice. Therefore we are going to work with samples. Our first generation of chromosomes will have a population size `POPSIZE` given by Slovin's formula. We remark that this formula is not recommendable except for special cases like ours (see Tejada and Punzalan [21] for details). Of course, the members of this generation are produced at random. In C++11 we use the method `uniform_int_distribution` of the class `random_device`. If we want the first generation to represent all chromosomes with a margin of error $\epsilon = 10\%$ then

$$\mathtt{POPSIZE} = \frac{N}{1 + 0.1^2 N} \cong 100. \tag{4}$$

This was already expected because Slovin's formula gives a sample size close to ϵ^{-2} when N is very large.

Now the question is: how many times should a chromosome `c` be tested if it has Gene 4? First of all, it can happen that the robot tracks along a certain $G = (V, E)$ according to `c` and accesses *only* Genes 0, 1 and 2. An example is a square grid G where the robot starts on a corner with situation $\mathcal{DDN}$. Now, if `c`=44444444424414444244144140 the robot will encounter only four situations besides that previous one: $\mathcal{DNN}$, $\mathcal{DCN}$, $\mathcal{DDC}$ and $\mathcal{DCC}$ (we recall that it will stop as soon as all vertices have been cleaned). Hence it never accesses *any* Gene 4 in `c`.

Because of that we shall focus on *how* the trajectories are affected in the *presence* of Gene 4 regardless how frequently it appears in the chromosome. Since the robot takes at most four directions when it casts the dice we shall consider that $5 \cdot 4 = 20$ trajectories are enough to evaluate the fitness of such chromosomes. The factor 5 is justified in Yates et al. [22, p.734].

As mentioned in the Introduction, our fitness `f` of a chromosome `c` consists of an ordered pair (`nclnd`, `bttry`) where the 1st and 2nd coordinates indicate how many spots were cleaned and how many Joules remain in the battery, respectively. We recall that each vertex can be cleaned just once. Hence the GA strives to find chromosomes `c` that give `f(c)` = (`nv`, `bttry`), where `bttry` is as large as possible. The battery must start with 13.5 `nv` J, as explained in the Appendix.

5.6 Structure of the Code

As explained in Section 5.2 we get a graph $G = (V, E)$ as the output of `equalis`. This graph is stored in a typical Evolver-datafile `graph.fe`, where the FE-extension is explained in Brakke [20, Ch. 2]. In our case `graph.fe` consists of a list of vertices followed by a list of edges. Each line begins with an identification number `id`, followed by coordinates in the case of a vertex, and by two vertex id-numbers in the case of edges. The code consists of five CPP-files (`main`, `ios`, `run_ga`, `walk`, `graphs`) and also `prototypes.h`, this one to declare global variables, structures and functions.

It starts by invoking a function defined in `ios.cpp` as `readf`, which reads `graph.fe` and stores its data into the structures `vert[]` and `edge[]`, these ones defined as prototypes. Right afterwards, the 1st generation is created by `genpp()` defined in `run_ga.cpp`, and from that point on `ppscr()` performs a scrolling of the populations. That function is defined in `run_ga.cpp`, and it goes from the 1st to the *final generation*, which we establish as the one that has just achieved at least 60% of members able to make the robot visit all vertices.

While scrolling, `ppscr()` calls the subfunction `cft()`. This one computes fitness `f` by making the robot perform 20 walking sessions with each chromosome `c`, as mentioned in Section 5.5. There we have also explained that `f(c)` = (`nclnd`, `bttry`), but from the 20 sessions `cft()` will only return the best `f(c)`. Namely the greatest `bttry` for the greatest `nclnd`. Hence the final generation will always have at least 60% of `nclnd`-values equal to `nv`.

The subprogram `walk.cpp` includes `walk()` and also 7 subfunctions, which altogether correspond to the implementation of the strategies explained in Section 5.3. Each new generation is created in `ppscr()` after crossover and mutation performed by `xov()` and `mut()`, respectively and in this order. Both belong to `run_ga.cpp` and now we give some details about them.

In any generation the members are ordered from best-to-worst fitness right before invoking `xov()`. This function performs a fitness-based roulette wheel selection that works with a very simple strategy. In the ordered generation suppose the position $1 + x\cdot$`POPSIZE` was chosen, where $x \in [0, 1 - 1/$`POPSIZE`$]$. Take a positive power $p \in \mathbb{N}$ and change this position as $x \leftarrow x^p$. For instance, if $p = 2$ the 51st position is replaced with the 26th, the 41st with the 17th, and so on. Of course, in this case `POPSIZE` $= 100$ will make positions 1 to 10 all turn 1, 11 to 15 all turn 2, and so on. Hence one should take a relatively small p, otherwise the roulette wheel approaches a purely elitist selection (non-recommendable in GA theory, see Holland [23] and Goldberg [24] for details). We have fixed $p = 2$ and also checked $p = 1$ to see that, in this case, the members with `nclnd` = `nv` are always between 5% and 27% of the population in each generation. Namely, $p = 1$ will never result in a final generation.

We have fixed 10% of crossover rate, namely 20 children always replace 20 members of each generation. These are also chosen by a fitness-based survivor selection whose strategy is quite similar to the one we have just mentioned. Now take $x \leftarrow x^p(2 - x)^p$ in order to increase the chances of replacing the worst chromosomes, again with $p = 2$ for the same reason. Mutation applies to 5% of the chromosomes in the extended notation with 26 essential entries. Since 12 genes never count for the short notation, then in practice the mutation rate remains around 2.5% (very small, as recommendable in GA theory, see Holland [23] and Goldberg [24] for details).

5.7 Running the Code

As explained in Section 5.3 a success happens whenever we find `c` with `f(c)` = (`nv`, `bttry`). As soon as the successes in a population reach at least 60% the program draws a cleaning session of the best chromosome and saves the final generation in a file named `example.txt`, where the chromosomes are listed in the best-to-worst order of fitness.

By opening `example.txt` the user will see a list of `POPSIZE` items like the one shown in Table 4.

In Table 4 the 1st, 2nd, 3rd and 4th columns list the chromosomes, `nclnd`, `bttry` and `dev`, respectively. The variable `dev` is the deviation that corresponds to the

Table 4 A typical example file

22 – 21 – – – –21 – 42 – – – –40 – 11 – 000	441	3649	-51
44 – 30 – – – –44 – 41 – – – –12 – 10 – 000	441	3236	-133
22 – 24 – – – –11 – 00 – – – –11 – 24 – 110	441	3045	-247
	⋮		
41 – 13 – – – –10 – 31 – – – –01 – 22 – 010	219	3002	-162
10 – 10 – – – –20 – 11 – – – –11 – 10 – 000	194	2934	-133

specific amount of energy consumed at rewinding (see $\mathcal{E}_2$ in Section 5.3). The quantity `bttry` is already debited of `dev` but *just once*. As explained in Section 5.3, this corresponds to the case $(2 + \cos\theta)$J where $\theta \equiv 180°$.

In general `example.txt` has only *non-deterministic* chromosomes, as named in Section 5.4. A few ones without Gene 4 can appear, and also a couple of repeated chromosomes, but always with distinct `f(c)`.

The user can choose a candidate, normally the one on the 1st line, and run again the program by resetting `POPSIZE` to 1. In this case `example.txt` is overwritten with the best score of 20 cleaning sessions for that single chromosome. Vertex id-numbers are then stored in a file `test.txt` to describe the sequence S mentioned in Section 5.3. If the user wants to record the final generation, then a separate copy of the initial `example.txt` must be kept in advance.

6 Results

As mentioned in Section 4, we took Robby's example from Mitchell [13, Ch.9] as an inspiration for this present work. But in Mitchell [13, Ch.9] one sees that Robby can score at most 500 points. The author used her GA and obtained 483 in average after 10 thousand tries. Now, our Scrubbrushy gets an almost grid graph $G = (V, E)$ to clean all vertices and save as much battery as possible.

In Section 5.4 we explained why Scrubbrushy follows a sequence S of vertices with $|S| \leq 1.5|V|$. Formula (4) in Section 5.5 justifies the choice `POPSIZE` =100. But differently from Robby's case, the maximum amount of energy that Scrubbrushy can save is unknown. Surely it spends at least $|V|$ J, and in the Appendix we show why it starts with $13.5|V|$ J. However, the optimal value would only be known if we could check all the $\mathcal{O}(3^{|V|})$ trajectories, as pointed out in Section 5.5, which is of course intractable. We recall that $|V| =$ `nv`, and the double notation is explained in Section 5.3.

Hence we must rely on some experimental values. For instance, the 60% mentioned in Section 5.6 was always attained by our GA for the grids $G = (V, E)$ inspected until now. By changing the code for 65%, 70% or even more we also get convergence. But on the one hand the best chromosomes `c` never attain fitness values `f(c)` = (`nv`, `bttry`) with higher `bttry` than in the 60% case. On the other hand `bttry` of the best `c` starts falling for 55%, 50% and lower values.

That is why we chose 60%, and with this value we ran the code 100 times for each grid. The best `c` had values of `bttry` that did not differ very much, even considering the deviation `dev` explained in Section 5.7. However, for each grid we took the best values of `bttry` to discuss in the next subsections.

Of course, any GA is typically run much more than just 100 times. However, we work with values that will be adjusted after tests with a toy robot in a maquette, as mentioned in Section 3. For instance, take the formula $(2 + \cos\theta)$J of Section 5.3. There we explain that our GA uses the integer approximations, and this is one of the reasons that speed up our code: we do not work with floating-point operations.

Such adjustments will be resumed with a real size robot immersed in a reflection pool. Then we shall have to consider the irregularities of the paving, and also slipping caused by wind and water movement. Our GA code is going to include parameters and values that correspond to the real approach of a robot in a reflection pool. Once we have adjusted the code, then it will be run multiple times in order to get better and compelling results.

For the time being, still without the aforementioned adjustments, the code runs relatively fast in our platform: 4GB of RAM, microprocessor Intel Core i5 3.2GHz, operating system Linux Ubuntu 14.04 and NetBeans 8.0.2.

The next subsections describe experiments with three different grids. They also show comparisons of the GA performance, from which the best chromosomes were selected. Three is of course a small number of grids but our GA is relatively fast. Namely, for any given grid the user can run our GA and then compare its best offsprings with the performance of **chr**.

6.1 First Experiment

For the graph of Fig. 3c execution time varies between 1min28s and 3min35s in our platform. As explained in the

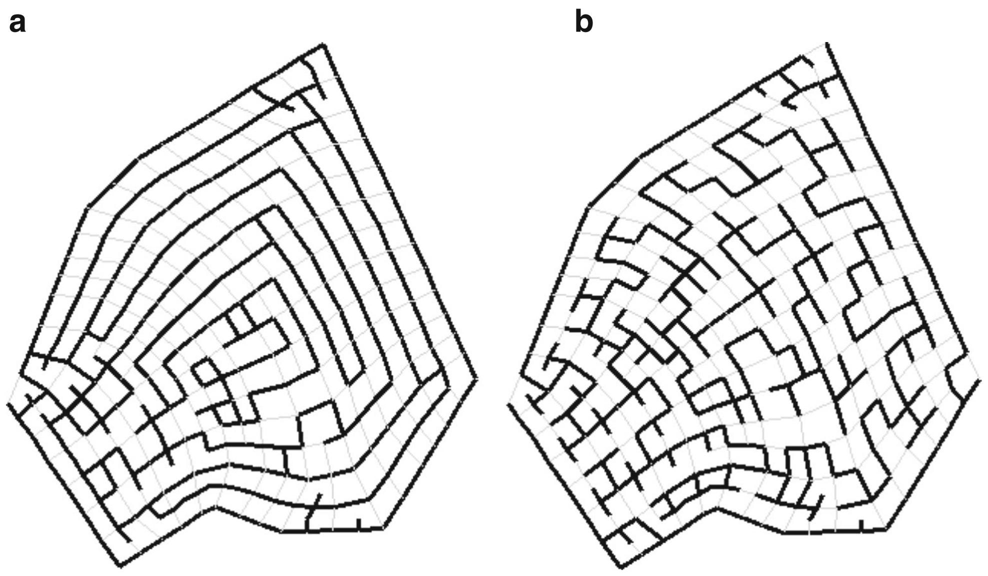

Fig. 9 Graphical outputs of trj7a (left) and trj7b (right) for Fig. 3c (cf. Fig 8a)

Appendix the battery should start with 13.5 `nv` J. In our case `nv`= 289 ≅ 300, hence we shall take 4050J.

Of course, any deterministic chromosome has constant fitness and deviation, which are (289, 2201) and -288 for **chr** in this case. Namely, **chr** makes the robot visit all vertices and the battery will end up with an amount of energy between 1973J and 2201J.

However, in this experiment another two chromosomes were technically as good as **chr**, namely

$$\text{chr7a} = 12 - 13 - - - - 23 - 40 - - - - 14 - 24 - 140,$$
$$\text{chr7b} = 32 - 11 - - - - 10 - 00 - - - - 22 - 11 - 000.$$

With chr7a the robot performed a trajectory named trj7a with `bttry`=2145 that saved at least 1924. See Fig. 9a for an illustration of trj7a. It is now stored in a test-file to be checked for a toy robot in a maquette.

Now notice that chr7b is deterministic, whence it makes the robot always perform the same trajectory trj7b. In this case `bttry`=1991 and `dev`=-296, so that it could outperform **chr** of 18J. But this is a negligible difference of 1%, and it holds only for the improbable case $\theta \equiv 180°$ (see Sections 5.3 and 5.7). In the next subsection we are going to see that these two chromosomes have a poor performance compared with another one that is non-deterministic.

6.2 Second Experiment

For the graph of Fig. 5c execution time varies between 1min4s and 1min48s in our platform. Now the battery starts with 6100J because `nv`= 441 ≅ 450. Unlike **chr** we are going to see that chr7b now makes the robot take a trajectory trj8a that omits *one* $v \in V$. But since chr7b was comparable to **chr** in the 1st experiment we shall always include it in

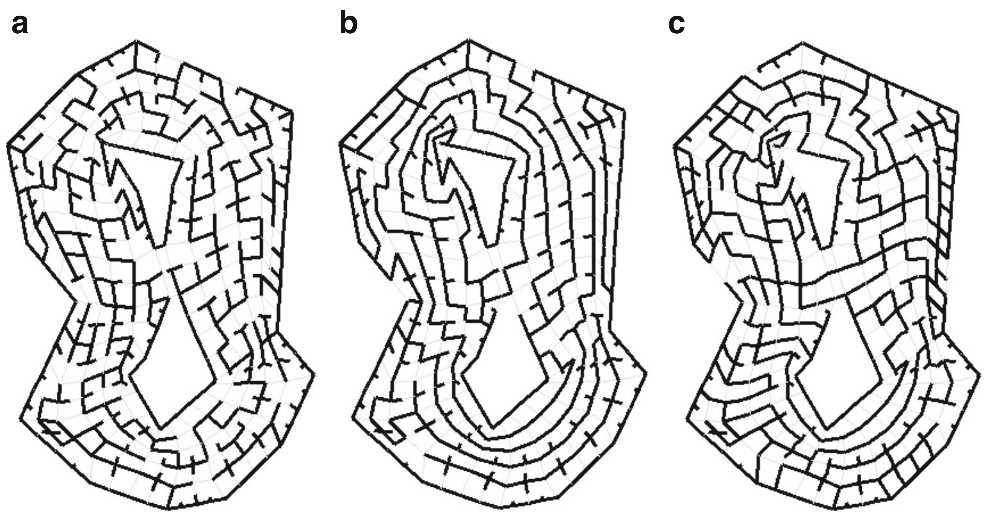

Fig. 10 Graphical outputs of trj8a (left), trj8b (centre) and trj8c (right) for Fig. 5c (cf. Fig 8b)

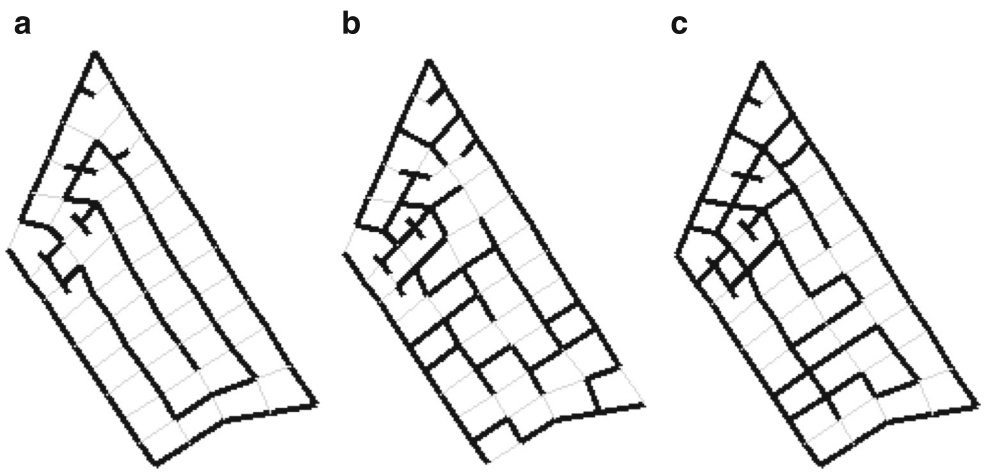

Fig. 11 Graphical outputs for **chr** (left), chr7b (centre) and trj9c (right)

our tests. The new performances are 3081(-440)J and 2070 (-661)J for **chr** and ch7b, respectively. However, this time **chr** has scored a technical draw with

$$\text{chr8b} = 12-13----22-20----14-12-400,$$

which was able to produce a trajectory trj8b with 3100 (-285)J. See Figs. 10a-b and 8b for the graphical outputs. We also included trj8c of chr 7a, which visited all vertices but with a performance of 2880(-291)J.

6.3 Third Experiment

Now we take a subgraph of Fig. 3c with `nv` = 72, for which execution time ranges from 10s to 18s and the battery starts with 1020J. The performances of **chr**, chr7b and a new chromosome

$$\text{chr9a} = 44-43----22-42----11-02-100$$

were 555(-71)J, 495(-81)J and 590(-28)J, respectively. This last one was achieved for a trajectory named trj9c. See Fig. 11 for their graphical outputs.

7 Conclusions

As mentioned in the Introduction, one of the greatest challenges of projecting an autonomous robot is the navigation modulus. This one includes the planning of near-optimal routes that enable the robot to move all through the area of interest by detouring from any obstacle.

In this work we presented a Genetic Algorithm that yields near-optimal chromosomes in the following sense: they represent a sequence of actions (genes) for the robot to clean an arbitrary reflection pool as efficiently as possible. For the Coverage Path Planning some works in the literature consider obtaining a near-optimal sequence S of vertices. The robot will follow S, as in the case of Giardini and Kalmár-Nagy [18]. Differently from these works we search for a sequence of actions that the robot will follow to optimise the cleaning of a reflection pool.

As an input the robot gets a graph $G = (V, E)$ where V and E are the sets of vertices and edges, respectively. Moreover, in G almost any angle from a pair of incident edges is either right or straight within a tolerance margin. Robotic displacements are quite simpler to program for a graph with these characteristics, and in our previous work [8] we showed how to obtain such G for arbitrary pool contours.

Our next work will include simulations with a maquette of a pool and a toy robot, which we can build with a kit like *Lego NXT*. The toy robot will follow a trajectory, or even a deterministic chromosome, as exemplified by our experiments in Section 6. We shall also implement trajectories obtained with other strategies of Coverage Path Planning in the literature to compare them with ours. Finally, as a second future work we shall implement a real size robot that will actually clean reflection pools. This part includes computer vision to aid in the definition of routes and in the identification of sites that have already been cleaned.

Acknowledgements The authors thank Prof José Artur Quilici Gonzalez, Federal University of ABC, for his assistance with the state of the art and also for valuable discussions. We are grateful with the referees' suggestions, which helped improve our work substantially.

Appendix

Here we prove some important results previously cited in the text.

Preliminaries

In Section 5.2 we presented the equalisation procedure that results in $G = (V, E)$. Each closed polygonal $v_1 \rightarrow v_2 \rightarrow$

$\cdots \rightarrow v_k \rightarrow v_1 =: \mathbb{P} \subset E$ such that the interior of $\mathbb{P}$ has at most one element $(v, e) \in V \times E$ is called a *cell* of G. Without going into details, the equalisation will always produce G with the following properties $\mathcal{P}_1 - \mathcal{P}_3$:

$\mathcal{P}_1$: $4 \leq k \leq 6$.
$\mathcal{P}_2$: $\forall\, v \in V \mid deg(v) = 1$, the unique $w \in V \mid \overline{vw} = e \in E \Rightarrow deg(w) = 4$.
$\mathcal{P}_3$: if v, w and e are as in $\mathcal{P}_2$, then $\exists \{w_1, \ldots w_4\} \subset V \mid$ the polygonal $w \rightarrow w_1 \rightarrow \ldots w_4 \rightarrow w$ has only v and e in its interior.

In the next subsection we study the special chromosome **chr**.

Visiting All Vertices

At the end of Section 5.3 we introduced two important exceptions $\mathcal{E}_1$ and $\mathcal{E}_2$, which together with Table 2 will guarantee that **chr** makes the robot visit all vertices in V. The proof is by contradiction. If there is $v \in V$ that the robot never accesses, then take $w \in V$ such that $e = \overline{vw} \in E$. Hence w is also never accessed, because $\mathcal{E}_2$ guarantees that trajectories are rewound. Since G is connected then we apply the same argument successive times until concluding that the robot never accesses any element of V, which is a contradiction. Therefore, the robot will indeed visit all vertices with **chr**.

But this chromosome can be quite expensive. In Fig. 12 we show $G = (V, E)$ with $|V| = 76$, and the robot follows **chr** by starting at 1.

The robot surrounds the outer contour and reaches vertex 52 exactly at the 52nd step. Then it goes to the boundary of the shaded region, an obstacle that the robot finally surrounds completely at the 80th step. Namely, the robot repeated $80 - 76 = 4$ vertices until now. Vertex number 56 is the last one the robot visits without repetition, because the 57th step is again vertex 55. The robot has not cleaned many vertices yet, so it will rewind to vertex 47 at the 112th step (the next one is indicated as 113). The robot will finish cleaning the whole pool at the 173rd step, which is precisely the vertex between numbers 1 and 2. Hence it

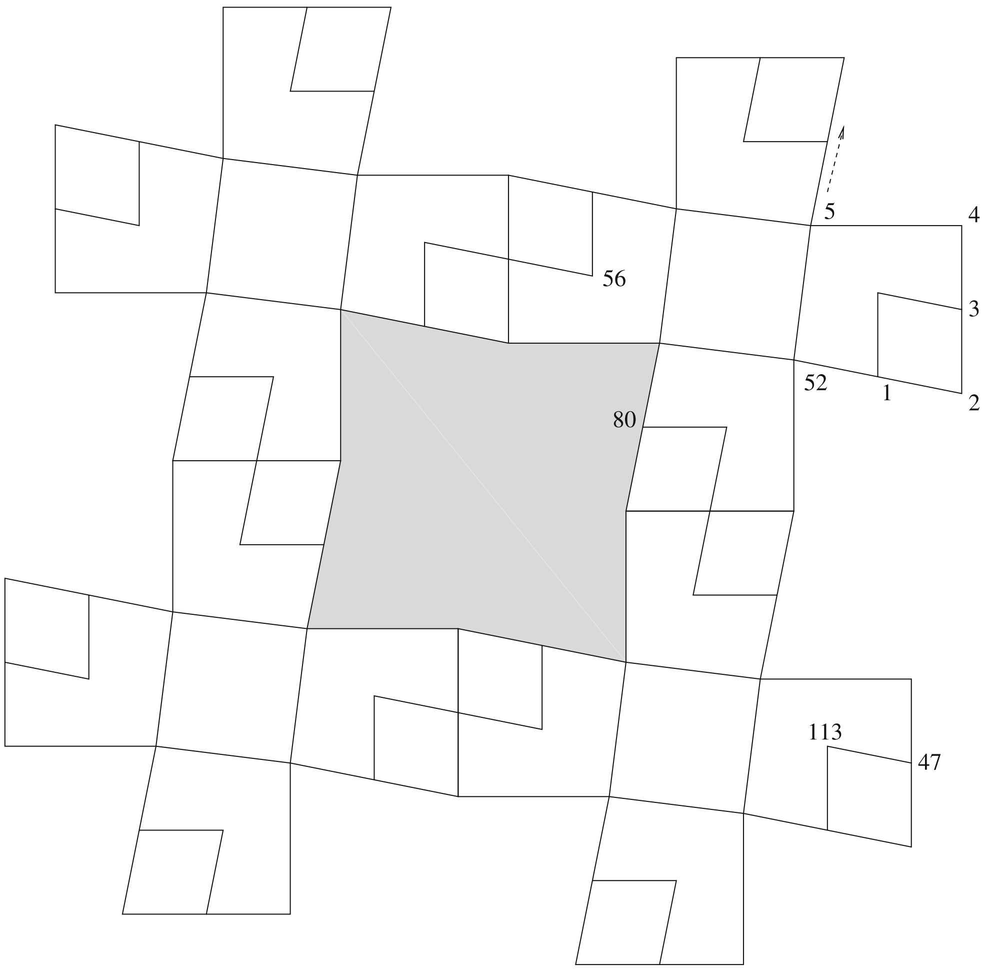

Fig. 12 Example of $|S| > 2.27\,\texttt{nv}$

takes a number of steps that is $173/76 \cong 2.2763$ times $\mathtt{nv}$, where $\mathtt{nv} = 76$.

Optimal Trajectories

Now we prove that with an optimal trajectory the robot visits all vertices with $|S| < 1.25\,\mathtt{nv}$. Fig. 13 depicts all topological kinds of cells that appear in $G = (V, E)$ due to the equalisation procedure explained in Section 5.2. One of them is presented as a double-cell, two indicate (v, e) contained in their interior. Black and white bullets fit together, and one begins with either 4 or 6 vertices to re-construct G by starting from one of its cells.

In Fig. 13 the top left cell needs 2 steps to be cleaned. If we do not count the double-cell, then cleaning the others will take 4 to 5 steps each. Hence, in the worst case we need a number m of steps that is 5 times the number N of cells to clean all vertices.

If we re-construct G by starting from one cell then $4(N-1)$ vertices will be added to the initial 6 in the worst case, so that $\mathtt{nv} = 4N + 2$. Hence

$$m \leq 5N = \frac{5}{4} \cdot (4N+2) - \frac{5}{2} < 1.25\,\mathtt{nv}. \quad (5)$$

However, this optimal trajectory can lead to a high battery consumption. This is because each cell with its interior (v, e) is completely cleaned before the robot goes to the next one. According to our scores the robot spends at most $(2+4+3)J = 9$J to go, clean and return from a spot. Hence we set our battery to start with at least $9 \cdot 1.5 \cdot \mathtt{nv}$ J=13.5 $\mathtt{nv}$ J, where the factor 1.5 is explained in Section 5.4. But this starting value will be enough *providing* $\mathtt{dev} >$ -13.5 $\mathtt{nv}$, where $\mathtt{dev}$ is defined in Section 5.7.

References

1. Abramson, S., Levin, A., Levin, S., Gur, D.: Window cleaning robot. US Patent App. 15/370,209 (2016)
2. Roumagnac, M.: Pool cleaning robot. US Patent 7,805,792 (2010)
3. Garti, E.: Pool cleaning robot. US Patent App. 11/896,611 (2009)
4. Baek, B.K.: Vacuum cleaner robot. US Patent App. 29/294,635 (2009)
5. Zhang, Y., Zhang, W., Bi, J.: Recent Patents on Mechanical Engineering **10**(1), 30 (2017)
6. Fabris, A.E., Nascimento, M.Z.d., Batista, V.R.: Revista SODEBRAS (9), 168 (2014)
7. Wang, C., Tang, Y., Zou, X., SiTu, W., Feng, W.: Optik-International Journal for Light and Electron Optics **131**, 626 (2017)
8. Zampirolli, F.d.A., Quilici-Gonzalez, J.A., Ramos Batista, V.: In: 6th IASTED International Conference on Modelling, Simulation and Identification, vol. 1, pp. 25–31 (2016)
9. Pichon, P., Deloche, R., Blanc-Tailleur, P.: Swimming pool cleaning apparatus with extractable filtration device. US Patent 9,657,488 (2017)
10. Renaud, B.J., Renigar, S.D., Hayes, G.M., Osuna, O.E.: Pool cleaning device with wheel drive assemblies. US Patent 9,677,294 (2017)
11. van der Meijden, H.J., Moore, M.E., Harbottle, B.D., Klimas, D.A., Bauckman, M.J.: Automatic pool cleaners and components thereof. US Patent 9,611,668 (2017)
12. Corke, P.: Robotics, vision and control. Springer, Berlin (2013)
13. Mitchell, M.: Complexity: a guided tour. Oxford University Press, Oxford (2009)
14. Yang, S.X., Luo, C.: IEEE Trans. Syst. Man Cybern B (Cybernetics) **34**(1), 718 (2004)

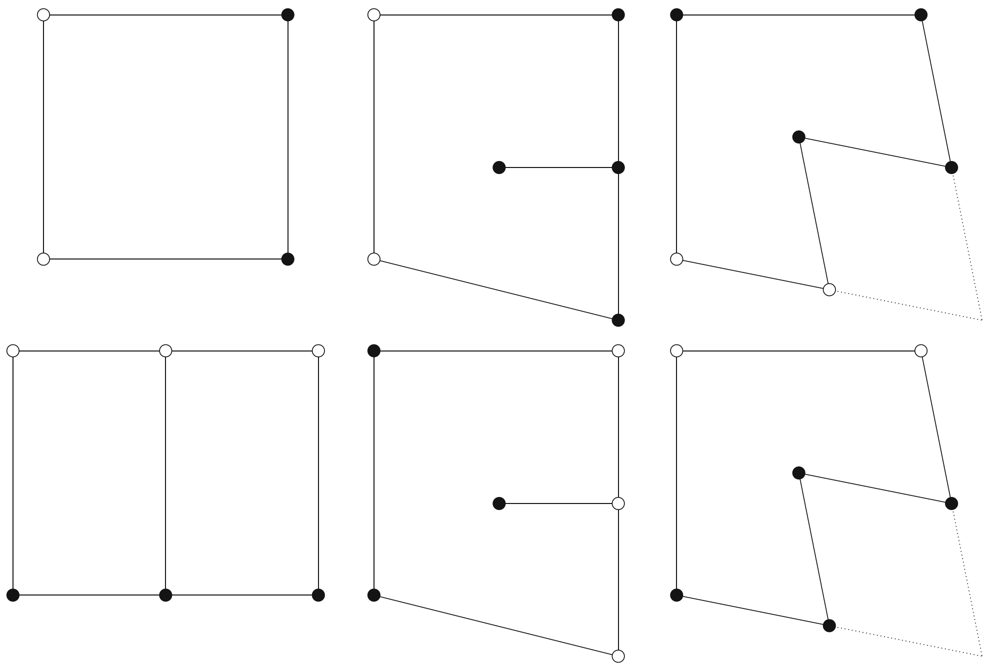
Fig. 13 Kinds of cells and the way they fit together to form $G = (V, E)$

15. Luo, C., Yang, S.X.: IEEE Trans. Neural Netw. **19**(7), 1279 (2008)
16. Nedjati, A., Izbirak, G., Vizvari, B.: J. Arkat, Robotics **5**(4), 26 (2016)
17. Ersson, T., Hu, X.: In: Intelligent Robots and Systems, 2001. Proceedings. IEEE/RSJ International Conference on, vol. 2 (IEEE 2001), pp. 858–864 (2001)
18. Giardini, G., Kalmár-Nagy, T.: Math. Probl. Eng. **2011** (2011)
19. Driscoll, T.A., Trefethen, L.N.: Schwarz-christoffel mapping, vol. 8. Cambridge University Press, Cambridge (2002)
20. Brakke, K.: Susquehanna University. Retrieved July 2013 from http://www.susqu.edu/brakke/evolver/evolver.html (2013)
21. Tejada, J.J., Punzalan, J.R.B.: The philippine statistician **61**(1), 129 (2012)
22. Yates, D., Moore, M., McCabe, G.: The practice of statistics. W.H. Freeman, New York (1999)
23. Holland, J.H.: Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence (University of Michigan press Ann Arbor) (1975)
24. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning (Addison-Wesley) (1989)

Valério Ramos Batista graduated in Computer Engineering at the Technological Institute of Aeronautics in SJ Campos, Brazil, Dec 1993. He concluded his master's in mathematics at the University of São Paulo, Brazil, Sep 1996, and his PhD in Mathematics at the University of Bonn, Germany, Jul 2000. Nowadays he works as a full professor in Computer Science at the Federal University of ABC, St André, Brazil. Prof. Ramos Batista was awarded the Scholarship in Research Productivity by the Brazilian National Council for Scientific and Technological Development, from Mar 2008 to Feb 2011. His main research areas are Computational Geometry and Image Processing.

Francisco de Assis Zampirolli graduated in mathematics at the Federal University of Espírito Santo in Vitória, Brazil, Dec 1992. He concluded his master's in mathematics at the University of São Paulo, Brazil, Sep 1997, and his Ph.D. in computer engineering at UNICAMP, Campinas, Brazil, Jun 2003. Nowadays he works as a full professor in computer modelling at the Federal University of ABC, St André, Brazil. His main research area is Image Analysis.