Computational Topology - group project

The Evasion Problem

Andrej Jočić
Matija Ojo

January 4, 2024

1 Introduction

The evasion problem asks if there is a way to move in an area covered with moving sensors while avoiding detection. We assume the sensors follow predefined paths and their movement is periodic. This means that the positions and movement directions of the sensors (and thus the area they cover) repeats itself after some global period, which is often the case with real sensor configurations. Furthermore, we assume the area in question is a subspace of the plane $\mathbb{R}^2$, however this is not necessary, since the method presented here can be trivially generalized to higher dimensions as well.

Aside from validating sensor configurations, this can be used to determine the minimal number of sensors (and their movement paths) needed to guarantee detection of an intruder. Clearly this could be achieved by covering the entire area with sensors at all times, but we can cut costs if we only want to make sure the intruder is detected *at some point* during the sensor movement cycle.

To ease implementation at the cost of some generality, we make the following simplifying assumptions:

- the intruder's movement speed is unbounded,
- the area in question is a rectangular grid,
- the paths along which the sensors move are straight line segments,
- all sensors move at the same speed of one unit of distance per one unit of time, and
- each sensor covers a square area two units of distance across oriented parallel to the edges of the room they protect.

2 Methods

By considering the unobserved (free) area at each point in time $t \in \mathbb{R}$ (a two-dimensional planar slice), we obtain the three-dimensional *free complex* F embedded in space-time. We are only interested in the times up to the (global) sensor network period p, since the object simply repeats itself after p. Therefore, the complex we obtain is similar to a torus, where the first point in time is "glued" to the time at $t = p$. One way to obtain the evasion paths would be to compute $H_1(F, \mathbb{Z})$ and check which generators represent valid cycles/paths (i.e. cycles going strictly forward in time). However, we decided to take a different approach: collapsing the free complex to a directed graph and computing cycles from its starting nodes. Before discussing that, we will first describe the data structures used to represent the input and the free complex (the code can be found in the project repository [2]).

2.1 Sensors

Since all the sensors have the same properties (area covered, speed of movement), they are uniquely defined by the `Path` which they follow. A `Path` is a sequence of `Position`s, where a `Position` is point in the plane (x, y). Since we want the sensors to move along straight line segments, two consecutive `Position`s need to either have equal x or equal y coordinates. The length of a `Path` is the sum of all the consecutive line segments that form it. Sensors move towards the next defined `Position` in the `Path`, and move towards the first `Position` when reaching the last one.

The `SensorNetwork` class is a collection of `Sensor`s along with the width and height of the rectangular area. `SensorNetwork` represents the entire configuration of the input, since the behaviour of the system is determined by the initial situation. The global period of the system (p) is computed as the least common multiple of all the sensor `Path`'s lengths.

2.2 Free complex

Due to the space discretization and axis-aligned sensor paths, the most suitable data structure for the free complex F is a cubical complex. We used the `CubicalComplex` from the gudhi library, which only allows axis-aligned cuboids.

We can simplify the complex construction by not taking any *wedge shapes* into account. A wedge shape is a situation where a cell in the grid was covered at time t, but is not covered at time $t+1$ or vice versa - not covered at t, but covered at $t+1$. This would result in cuboids that aren't axis-aligned, as seen in Figure 1. We can omit those shapes, because the intruder trying to avoid detection can either be in a particular cell or not - we do not allow partial occupation of a cell. Therefore, the valid positions for the intruder in the time interval $[t, t+1)$ are only those which are not covered at the two consecutive times t and $t+1$.

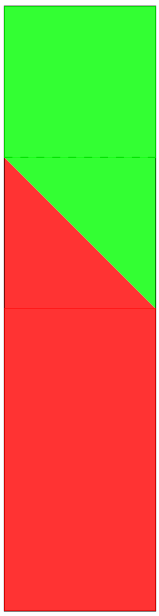

Figure 1: Top-down representation of a wedge shape in a cubical complex, with time increasing to the right. The covered area is shown in red and the free area is shown in green. In this case a cell was covered at time t but no longer covered at time $t+1$.

The covered planar slices are obtained by performing a union on all the covered areas by all sensors at each point in time. Covered area is simply a `Position` representing the top-left corner of a covered cell.

The cubical complex of dimensions $room\_width \times room\_height \times period$ is constructed by assigning the filtration value of 1 to free top-dimensional cells and 0 to covered cells. According to the reasoning described above, a cell in the complex is marked free *iff* the corresponding space is free at two consecutive time-steps t and $t+1$ (corresponding to 2 faces of the cube in F).

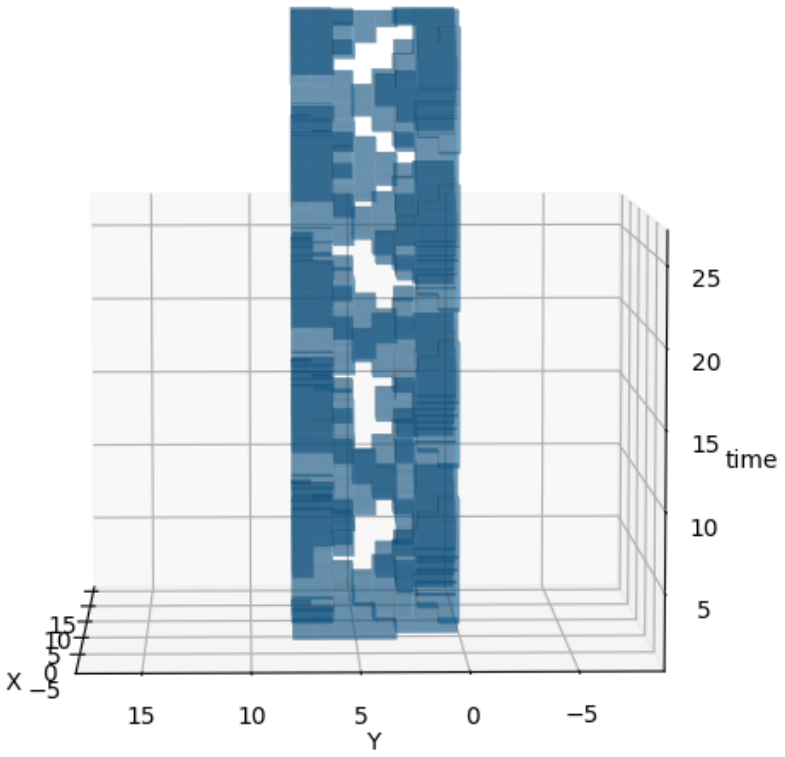

Figure 2: Free cubical complex of the example configuration (shown in Figure 5).

2.3 Computing evasion paths

Once the cubical complex F is obtained, we collapse it into a graph (1-dimensional complex) G in a way that preserves $H_1(F)$. A proper elementary collapse would have to preserve higher-dimensional homology as well, however we are only interested in $H_1(F)$. Incidentally, F shouldn't have any H_2 generators (enclosed caves) in the case of continuous sensor movement; this would imply there is something homeomorphic to a ball in the covered complex, meaning at some time t_0 one or more sensors appear out of thin air, and then disappear into a point at time $t_1 > t_0$.

The vertices of G correspond to connected components (by 4-connectivity) of F at each time interval $[t, t+1)$, $0 \leq t < p$. The idea is that an intruder with unbounded speed can move between any pair of points in a connected component of free space within a finite time interval without being detected. The vertices are labeled with (t, c) where c is the index of the connected component in this time slice, and they also hold information about the area they cover (so that we can obtain

concrete `Positions` after computing a cycle).

Edges in G allow the intruder to move forward in time, so an edge from (t, c) to $(t+1, c')$ is added *iff* the connected areas c and c' intersect (and thus the intruder can move from c to c' in one time step). If we wanted to strictly follow the homology approach to this problem, these edges would need to be undirected so that we end up with a simplicial complex on which to compute H_1. But in order to avoid having to filter out cycles that don't go forward in time, we kept track of the direction of edges in G.

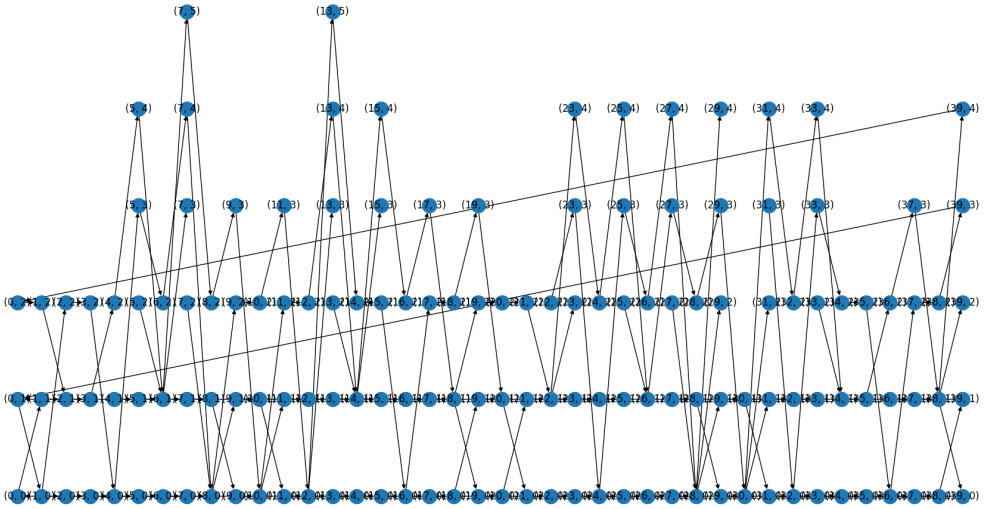

Figure 3: Evasion graph G of example sensor configuration (shown in Figure 5). Vertices are horizontally sorted by time. Note there are a few edges going seemingly backwards in time, but they are actually going forward in time, since the time is modulo the period.

Cycles of G are then obtained by performing a depth-first search (with depth limited to p) on the graph and checking whether the current node is equal to the starting node. The above construction guarantees that each cycle in G corresponds to a valid evasion path of length p.

Note that this method can be easily generalized to higher dimensions, since a higher-dimensional cubical complex can be collapsed to a graph in the same manner. For example, evasion paths in 3-dimensional space can be computed by dividing the 3-dimensional slices of a 4-dimensional complex into connected components, and connecting the vertices along the 4th dimension. Beyond that, we could perhaps model configuration spaces with even more dimensions.

3 Results

We tested the algorithm with several configurations. The first configuration is trivial: a single sensor moving in the counter-clockwise direction (Figure 4).

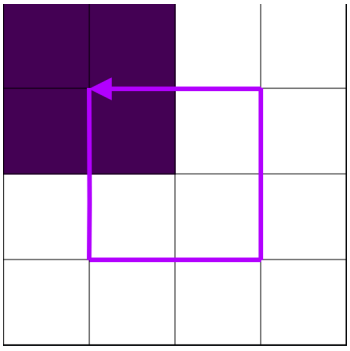

Figure 4: Counter-clockwise circular sensor in 4x4 room, period: 8

The algorithm correctly found 1 evasion path, an animation of which can be found here. The next configuration is the example from the instructions (Figure 5).

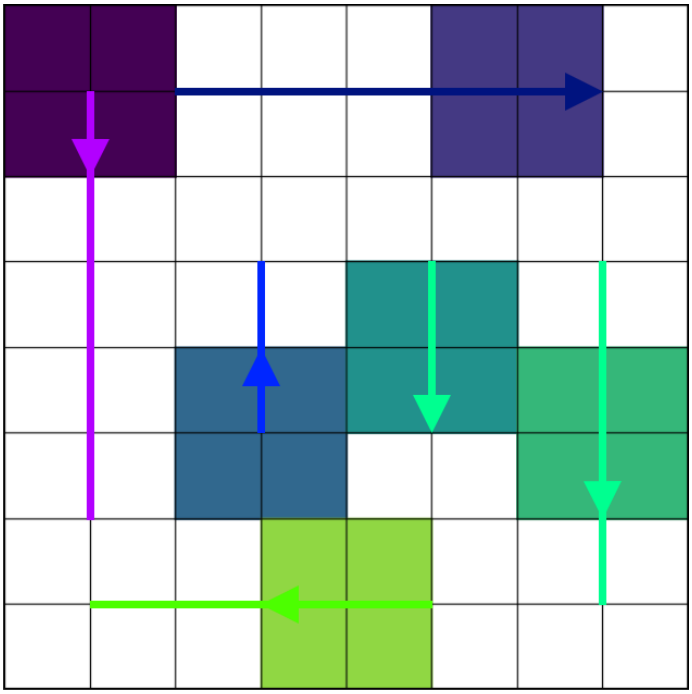

Figure 5: 6 sensors in 8x8 room, period: 40

The algorithm found 288 different evasion paths through components of free space; an animation of one such path can be found here.

Lastly, we tested the algorithm on random inputs, such that every point in the room is covered by at least one sensor at least part of the time. Here are some inputs, along with their animations:

- 10×13 room with 30 sensors, and a period of 80, where the algorithm found a single evasion path; an animation of the path can be found here.
- 12×18 room with 48 sensors, and a period of 56, where the algorithm found 2 evasion paths; an animation of one path can be found here.
- 12×24 room with 58 sensors, and a period of 84, where the algorithm found 68 evasion paths; an animation of one path can be found here.
- 20×18 room with 66 sensors, and a period of 80, where the algorithm found 247125 evasion paths; an animation of one path can be found here.

4 Discussion

We implemented evasion path detection by using the cubical complex structure, reducing it to a graph and finding the cycles there. This is equivalent to computing the $H_1(F, \mathbb{Z})$ on the free complex, however this way we did not need to handle edge cases relating to time travel.

We also implemented sensor paths formed of arbitrary line segments, and a room generator that creates areas of random sizes and sensor configurations. Lastly, we implemented animations of evasion paths which help to visually confirm the algorithm's correctness.

There is, however, a limitation to the algorithm that we only briefly discussed in Section 2.2. Since we only allow axis-aligned cuboids in the `CubicalComplex` (i.e. no diagonal edges), the algorithm is not able to detect *tight* evasions. By tight evasion we refer to the intruder's movement in free space that is covered by sensors in the previous and/or next integer time step. An intruder modeled as a single point in space *can* perform such an evasion, so we can just say we are modelling the intruder as a unit square that is dragging a cape behind it (which can still be briefly detected after the intruder has moved on).

This was done due to a limitation of the `CubicalComplex` in the gudhi library, so detection of tight evasions would require a different data structure for the free complex. Furthermore, the algorithm for collapsing the complex to a graph should be changed, such that the nodes of the graph are in time slices instead of time intervals, and the edges span through intervals.

Alternative approaches

Instead of collapsing the cubical complex to a graph, we could have computed H_1 generators of the free complex directly and then checked which ones correspond to valid evasion paths. To our knowledge, there is no software that can compute H_1 generators of a cubical complex, so we would first have to subdivide the cubical complex into a simplicial one (dividing squares into triangles and cubes into tetrahedra). Of course, we could also have implemented cubical homology computation ourselves, but that would require a lot of time and effort for something that would almost certainly take longer to compute evasion paths than the approach we took.

An alternative homology-based approach would be to represent the time-varying (2-dimensional) free space with a zig-zag filtration and derive evasion paths from zig-zag persistence information [1].

5 Bibliography

References

[1] Henry Adams and Gunnar Carlsson. "Evasion paths in mobile sensor networks". In: *The International Journal of Robotics Research* 34.1 (Nov. 2014), pp. 90–104. ISSN: 1741-3176. DOI: 10.1177/0278364914548051. URL: http://dx.doi.org/10.1177/0278364914548051.

[2] *GitHub repository: TDA_Evasion.* https://github.com/MOj0/TDA_Evasion.