Robot Planning - Target Rescue project

Luca Sartore

January 11, 2026

1 Introduction

This work aims to develop a planning module for a robot in a Target-Rescue scenario.

In this scenario the robot is placed into an environment with some obstacles, some victims to rescue and an exit point. The objective of the robot is to rescue as many victims as possible in a certain time window, and then reach the exit. Each victim has a value assigned, which the planner must take into consideration when selecting which victims to rescue. The robot is also constrained by the fact that it should move using only Dubins curves.

In order to achieve this task the robot's planning module is composed of three main submodules:

- **Planning graph module(2):**
  This module aims to read the map and convert it into a graph that can be used for planning. This module in particular has two versions: one based on a combinatorial approach, and another that uses a sampling-based strategy.
- **Trajectory planning module(3):**
  This module uses the Dubins motion equations to find a valid trajectory that connects two points of a specific graph with a specific orientation. The module also ensures that the path selected is feasible using collision detection.
- **Graph search module(4):**
  This module uses the previous two in order to find a good route that saves as many victims as possible. It is based on an A*-like algorithm.

2 Planning graph

The planning graph was implemented as an interface that converts a "map" (a simple data structure containing data about victims, obstacles, etc.) into a graph.

This is done in two possible ways: one is combinatorial (2.2) and the other is sampling-based (2.3). For the sampling-based approach, collision detection capabilities are also required (2.1).

2.1 Collision detection module

The collision detection module is designed to be extremely quick, and this is achieved by sacrificing a bit of accuracy.

This is done by splitting the map-space into a 1000×1000 grid. Obstacles and map borders are then "drawn" in the grid. Then a dilation [1] is applied, with a circular kernel with radius equal to the radius of the robot (plus a small margin).

In this way, to understand if a certain position is collision-free or not, we can simply check the grid square that is the best approximation of the position. This is an acceptable approximation (assuming that the grid is large enough) and, perhaps most importantly, is extremely fast, with every operation having $o(1)$ time complexity.

To check whether a path is collision-free or not, the path is approximated as a list of points with a certain resolution (in this work the resolution used was 5 points per unit of distance), and then each point is individually checked.

2.2 Combinatorial Planning graph

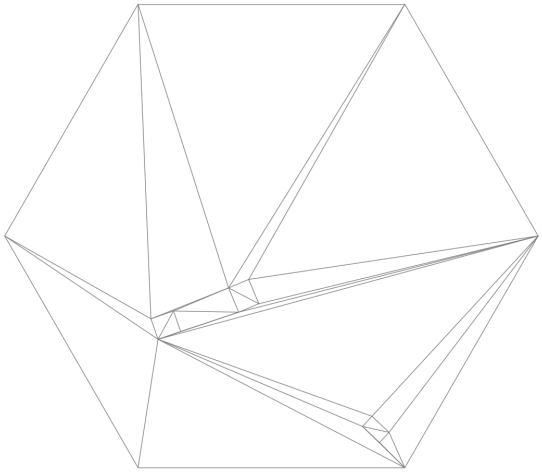
There are many different combinatorial strategies that can be adopted, "Triangular Decomposition", "Trapezoidal Decomposition" and "Approximate Cell Decomposition" just to name a few.

For this work, the triangle-based strategy was selected. The decision was motivated primarily by the observation that graphs obtained using triangles are usually more "smooth" than graphs obtained using trapezoids. Approximate cell decomposition was not selected due to the fact that it generally produces much larger graphs than the first two, which would result in higher computational costs.

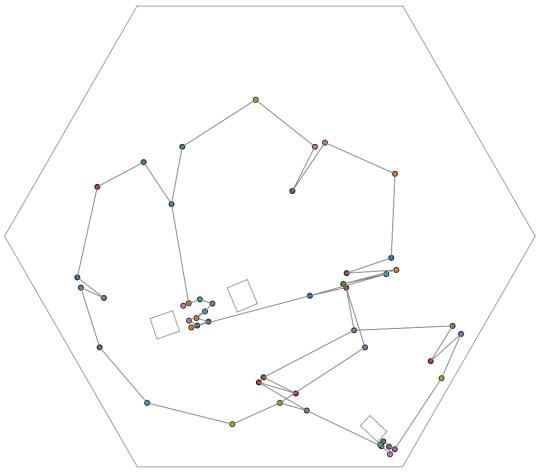
This was not necessarily the fastest option in terms of time complexity, however as we will see in the result section (5.1) the triangulation execution time is negligible when compared to the downstream steps.

2.2.1 Implementation details

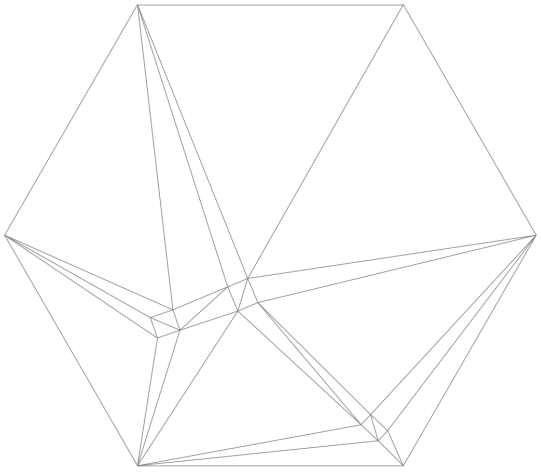
The implementation was done in a straightforward way. All edges were considered pairwise, and only



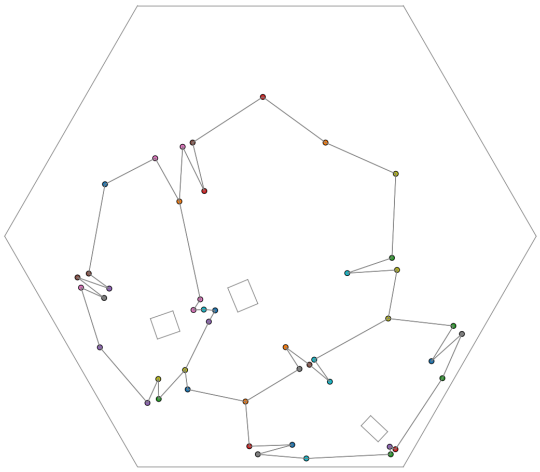
(a) Raw triangulation



(b) Graph generated by raw triangulation



(c) Flip triangulation



(d) Graph generated by flip triangulation

Figure 1: Comparison between different triangulation methods

considered valid if they did not intersect any obstacle. After this, a subset of the valid vertices is selected to form a valid triangulation.

Note that this procedure has a time complexity of $o(n^3)$ and there are more efficient implementations. The simpler implementation was selected as an improvement in speed here is negligible due to the downstream processing being much heavier.

The way arcs are selected to form a valid triangulation is not straightforward. A "simple" solution that simply picks a subset of arcs that don't intersect with each other can result in a triangulation that has many long and skinny rectangles, such as the one shown in figure 1a. This strange triangulation will then result in a graph that is hard to navigate, as we can observe in figure 1b

To solve this issue, the flip [2] algorithm was applied, resulting in a much better triangulation, and by extension, graph. The results can be seen in figure 1c and 1d

Another issue with the generated graph is that there are some "zig-zag" paths sometimes. Those are due to the fact that nodes are placed in the center of triangles as well as in the center of the sides. This particular path can be annoying for the robot to follow, and therefore an easy solution was to add "skip-ahead" connections to the graph. In simple terms, this means adding some direct connections from a node to all the nodes that can be reached in two steps. It is important to note that these new connections may not necessarily be collision-free, however, they will be checked for collision in the next step and be discarded if they are found to cause any issue. Overall, this addition made the path of the robot much smoother.

2.3 Sampling Planning graph

The sampling planning graph was obtained by first generating a random set of points (and ensuring that every generated point is a valid position of the robot by using the collision detection module 2.1)

Then the K-nearest neighbors of each node were selected, and nodes were added if and only if the direct path between the two nodes was judged to be valid by the collision detection system (2.1).

In this section a KD-tree was used to facilitate the retrieval of the K-nearest neighbors in an efficient manner.

Skip-connections were also added to this graph. They were not as important as they were in the combinatorial case, but they did provide a minor improvement to the graph. The output graph (before skip-connections were applied) can be seen in figure 2

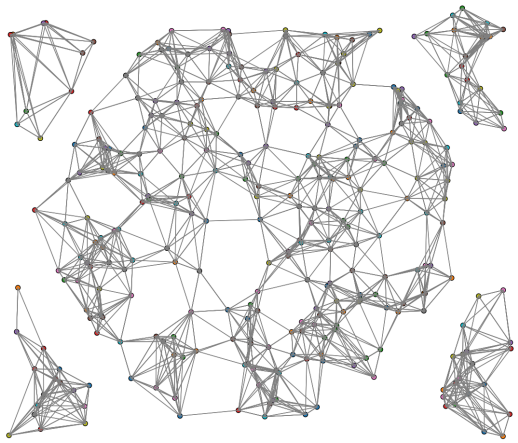


Figure 2: Graph obtained using sampling

3 Trajectory Planning

The trajectory planning is the simplest module of the project. It simply tests all the Dubins combinations (LSL, LSR, RSL, RSR, LRL, RLR) and picks the shortest one among those that are collision-free.

The collisions are checked using the collision detection module 2.1, and the time complexity is therefore $o(L)$ where L is the length of the trajectory.

The module generates 64 trajectories for each edge of the planning graph (given that it considers all the combinations of 8 start angles and 8 end angles). This is definitely one of the slowest parts of the program, however it is pretty easy to parallelize, and as we can see in the results section (5.1) the computation times are quite manageable.

4 Graph search module

The final step needed to complete the planner is the graph search module. This one takes the graph and the trajectories generated by the previous two modules and finds a trajectory that maximizes the value of the victims saved while ensuring it reaches the exit within a certain time limit.

The problem presented can be split into two separate problems: one is to figure out which victims to save, and what is the optimal pick-up order, and the other is to find the optimal path that picks up the victims in the decided order. We will therefore analyze the two problems separately in sections 4.1 and 4.2 respectively.

4.1 Optimal path finding

4.1.1 Problem definition

In this problem we are given a Graph with a starting point an ending point, as well as a set of nodes that we need to visit in a specific order before exiting the graph. There is also one specific constraint that makes it so that the edges we can use to exit a node depend on the edge that we used to reach the node (more specifically, they depend on the angle used when entering/exiting a node). This constraint is necessary to ensure that the Dubins paths selected are continuous at every node.

The goal is to reach the end in the shortest possible time (or distance) while still satisfying all the aforementioned constraints.

4.1.2 Implemented solution

The proposed solution can be seen as a version of A* with some minor modifications that allow it to satisfy the specific constraints of the problem.

The main modification w.r.t. A* is how we consider a node "visited": In the traditional A*, the state of nodes visited is simply a set containing the IDs of the nodes visited. However, in this modified version the set contains a triplet:

$$< node\_id, arriving\_angle, next\_objective >$$

This is done to take into account the fact that a node visited can be visited again if either it is visited by arriving with a different angle, or if the next objective in the list of nodes to visit has changed.

The heuristic cost is simply calculated by taking the euclidean distance between the current position and the next objective, plus the euclidean distance between the next objective and the second next objective all the way up to the exit of the graph.

A pseudo-code of the algorithm proposed can be seen below:

```
# graph data structure
graph = {}

# ordered list of nodes to visit
obj_list = [...]

# empty set with visited nodes
visited = {}

# priority queue
queue = {}
queue.push(
  priority = 0,
  value = {
    start_node,
    start_angle,
    0, # cost payed so far
    obj_list[0]
  }
)

while not queue.empty:
  # getting cost (so far),
  # the node id, the angle and
  # the next objective of the
  # first node in the queue
  next = queue.pop()
  node, angle, cost, obj = next

  # avoid revisiting nodes
  if {node, angle, obj} in visited:
    continue
  visited.insert({node, angle, obj})

  # found the solution
  if node == graph.exit:
    return

  # if we arrived at one objective
  # we will move to the next
  if node == obj:
    obj = next_obj(obj, obj_list)

  # iterate over all the edges
  # that start from node 'node'
  # and start with angle 'angle'
  for e in graph.edges(node, angle):
    # calculate heuristic estimated
    # cost to terminate the journey
    h = heuristic(e.next_node, obj)

    # find new cost paid so far
    new_cost = cost + e.edge_cost

    queue.push(
      priority = new_cost + h,
      value = {
        e.next_node,
        e.arriving_angle
        new_cost,
        obj
      }
    )
```

The described algorithm yields an optimal solution (w.r.t. the input graph, which may itself not be optimal), and the upper bound cost is quite manageable at $o(V \cdot E \cdot \log(E))$ where V is the number of objectives in the list and E is the number of edges in the graph.

4.1.3 Refinement process

In order to find a more accurate solution the first output of the program is subsequently refined twice. The refinement process consists in adding more angles of arrival for all the nodes that were part of the solution. The angles are placed in such a way that they are equally sampled in the space directly surrounding the angle chosen in the first solution. For example, if in the first iteration of a solution we pass through node A with an angle of 30 degrees, in the second iteration we would add angles 20, 25, 35, and 40 to node A's arriving angle possibilities. This step is repeated twice, yielding an average improvement of 0.98% and 0.12% for the first and second iteration respectively. It is obviously possible to continue this refinement past the second iteration, but the gains get progressively smaller.

4.2 Rescue order finding

4.2.1 Problem definition

In this problem we are given the list of victims with their positions on the map, as well as the value associated with each victim.

We also have a function that is able to tell us how long it would take to rescue a subset of the victims in a particular order (i.e., the program described in section 4.1)

The objective is to select which victims to save and in which order, so that the time required to save all the victims is below a certain threshold, and the value of the victims rescued is maximized.

4.2.2 Implemented solution

It is easy to see that the described problem requires exponential time in order to be solved optimally, as it is a sub-case of the Steiner TSP [3].

It is obviously possible to find some heuristics that find close-to-optimal solutions in a much faster time, however, since the scenario only had a small number of victims (4 in particular), we opted for a complete search of the solution-space in order to guarantee an optimal outcome. This was made possible thanks to the path-finding algorithm (4.1) being quite fast, and especially thanks to a multi-threaded implementation.

It is worth acknowledging that, if the conditions

were to change, nothing is preventing the use of a heuristic approach that could find "good enough" solutions in a reasonable time, even if the number of victims increases. However, with the scenario presented in the assignment, it makes more sense to use an approach that enumerates all possible solutions, as this guarantees the optimal solution while still having acceptable computational times.

5 Results

In this section we will show how the algorithm performs under various scenarios with a particular focus on noting and explaining the differences between the combinatorial and the sampling-based strategies for the creation of the planning graph.

5.1 Execution Time

In the table below we show the execution time of the two approaches. We also included a single core and a multi core run for each of the approaches tested.

All tests have been executed on a six-core twelve-thread Ryzen 5 5600. We would have preferred to run the tests using a more modern CPU, and especially one with more cores, to show the full potential of the parallel architecture chosen, but this was unfortunately unfeasible.

Different parts of the program were measured to understand where most of the time is spent. In particular, the measured parts were:

- **[Occ]:** Pre computing the occupation in the 1000×1000 grid used for collision avoidance (2.1).
- **[Gra]:** Building the planning graph (2.2 for combinatorial, 2.3 for sampling).
- **[Traj]:** Calculating all the trajectories using dubins (3)
- **[Sear]:** Searching the graph for the optimal path and victims order (4)
- **[Ref]:** Refinement of the best solutions (4.1.3)

	Execution Time [ms]			
	Sampling		Combinatorial	
	1Core	6Cores	1Core	6Cores
Occ	375	376	379	383
Gra	18	17	4	4
Traj	10881	1256	1229	142
Sear	12263	1925	2765	379
Ref	1585	197	1514	178
Tot	25126	3773	5892	1088

In the table we can note how combinatorial planning is much more efficient in terms of execution time. This is due to the fact that it generates a complete graph with a smaller number of nodes compared to the sampling based approach. However, this speed improvement comes at the expense of the quality of the trajectories, as we will see in the next section.

We can also note that the performance uplift we get from multithreading is excellent. We can see an improvement of approximately 6x when the process is compute-bound (such as, for example, the graph-search part of the sampling approach), and up to a 9x improvement when the process is memory-bound (such as, for example, the trajectory planning step). At first it may seem counterintuitive that a 6-core CPU manages to get a greater than 6x improvement in performance when using multithreading; however, this is simply due to the fact that the process is memory-bound, and the CPU is therefore able to take advantage of the 12 threads (in particular, the memory-bound part is doing collision detection by accessing the 1000×1000 grid 2.1).

5.2 Trajectory quality

In the two figures 3 and 4 we can see the same map solved by using a sampling-based and a combinatorial approach respectively. It is easy to see how the combinatorial approach is worse than the sampling one. This is intuitive, as the sampling one, although slower, is able to generate much larger graphs, which gives the graph solver more flexibility when it comes to finding a good path.

As we explained in section 4, the graph-search algorithm we presented is optimal with respect to the input graph. However, the two approaches proposed to generate a planning graph are not optimal, and therefore a better graph will produce a better path.

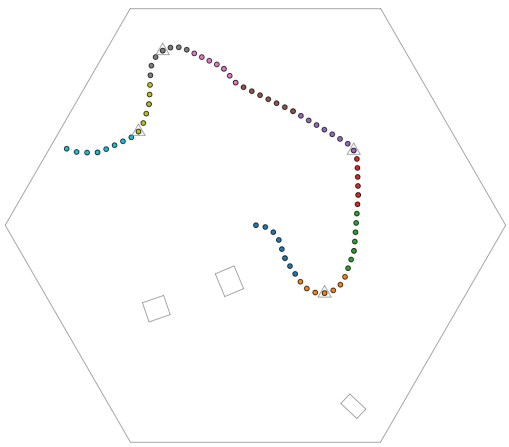


Figure 3: A path obtained using a sampling-based strategy

5.3 Collision avoidance

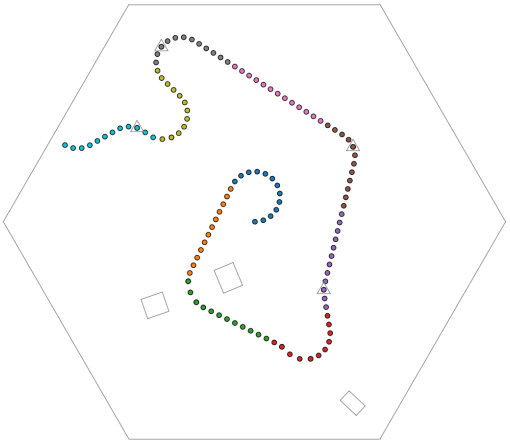


Figure 4: A path obtained using a combinatorial strategy

5.4 Collision avoidance

The proposed algorithm should guarantee that the found paths are collision-free (assuming that the robot radius is set correctly and with sufficient margin). Some crashes were still observed during the early phases of development, and after an investigation it was revealed that the controller did sometimes drift from the desired path by a significant margin (approximately 0.3 units of distance), especially when steering for prolonged periods of time. This was addressed by increasing the turning radius and adjusting the margin to account for controller drift. No other crashes were observed after these fixes.

5.5 Conclusions

In general, we can say that when comparing sampling-based and combinatorial strategies to build the planning graph, the sampling one is the better option for our scenario.

They are a bit slower than the other option when it comes to computation time, but they more than make up for it in the quality of the trajectory. For example, going back to the trajectories in picture 3 and 4, the sampling approach was 2.7 seconds slower in computation time, but resulted in a path that was 16 seconds faster, making the tradeoff totally worth it.

There are some variables that may affect this calculation; for example, increasing the speed of the robot will result in the combinatorial strategy being more competitive. On the other hand, the CPU used to run the tests is a bit outdated at this point, and I believe that simply using a more modern one could bring the execution time for the sampling strategy below one second.

5.6 More examples

In this section we will put more examples of the path selected by the robot in different scenarios. In particular there are 3 different scenarios, with varying numbers of victims and obstacles, and for each scenario there are two different solving methods (combinatorial and sampling) and two different time limits (120s and 30s). The 3 scenarios that we tested are the following:

- **Scenario A:** This is the small scenario with 5 obstacles and 4 victims, and can be seen in figure 5
- **Scenario B:** This is the medium scenario with 10 obstacles and 4 victims, and can be seen in figure 6
- **Scenario C:** This is the large scenario with 12 obstacles and 5 victims, and can be seen in figure 7

In general, we can note how when there is no time pressure there is not a big difference between the two strategies (they both usually manage to save all victims). However, when put under time pressure, there are some circumstances where the sampling-based approach is able to rescue more victims than the combinatorial-based one.

5.7 How to reproduce

The code is open-source and available on GitHub.
The parameters of the test can be configured from:
rescue_planner|src|utit|constants.hpp
And the test can be launched using:
roslaunch rescue_planner main.launch

6 Acknowledgements

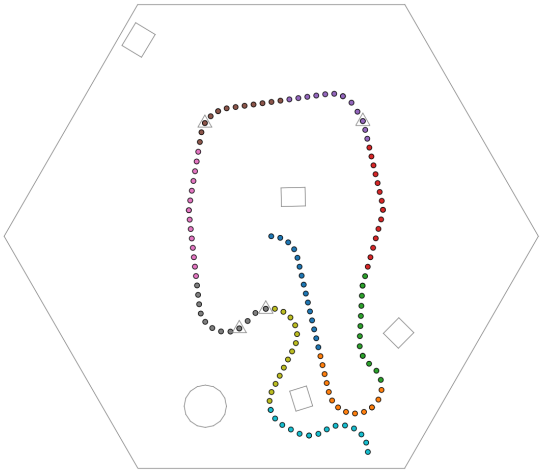
The code has been written entirely by me except for the part to generate charts (that is AI-generated) and the code for KD-Trees (taken from this open source library). The report has been written entirely by me, and spell-checked using AI.

References

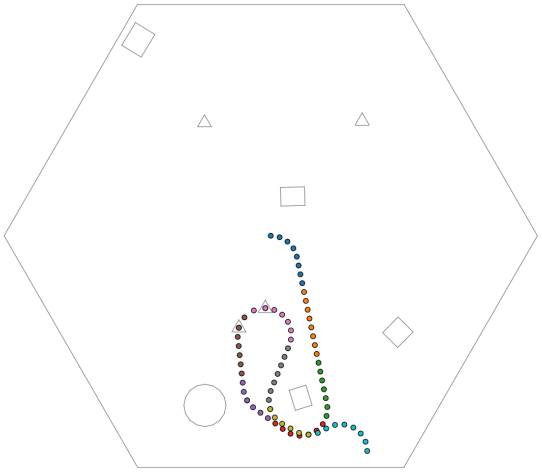
[1] OpenCV - Tutorial erosion and dilation

[2] Y. Jiang, Y.-t. Liu, and F. Zhang (2010) *An efficient algorithm for constructing Delaunay triangulation*

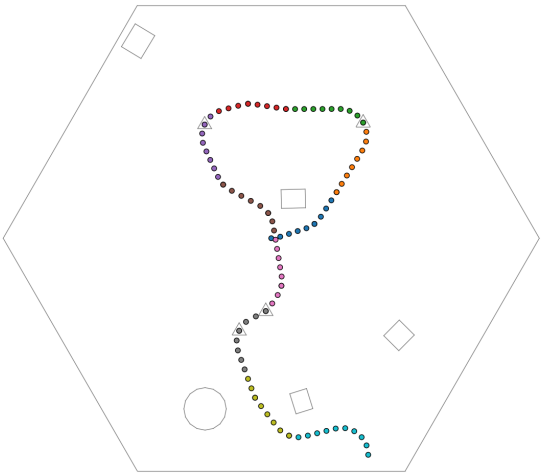
[3] Wikipedia - Steiner travelling salesman problem



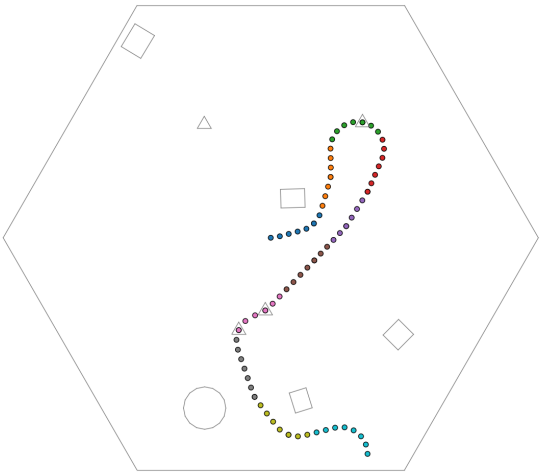
(a) Combinatorial, Time limit = 120s



(b) Combinatorial, Time limit = 30s

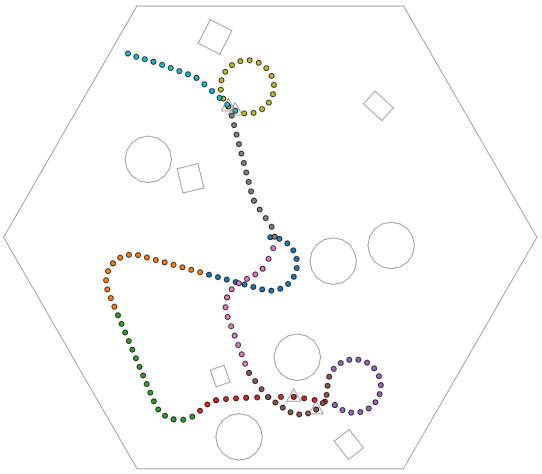


(c) Sampling, Time limit = 120s

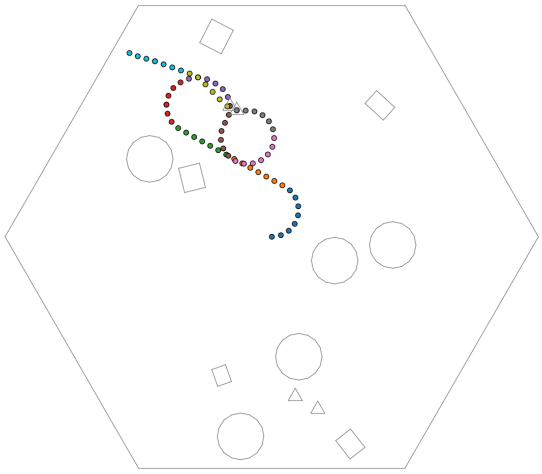


(d) Sampling, Time limit = 30s

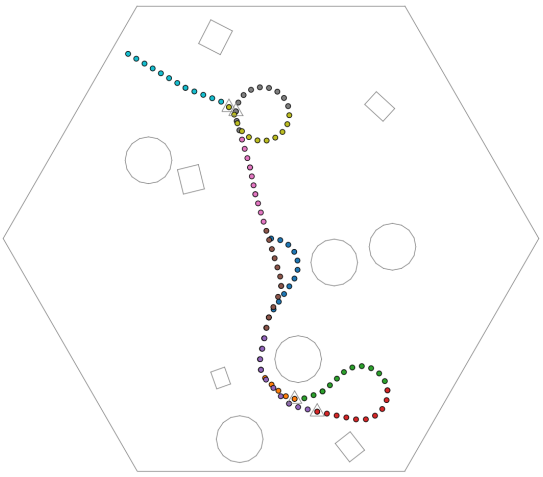
Figure 5: scenario A



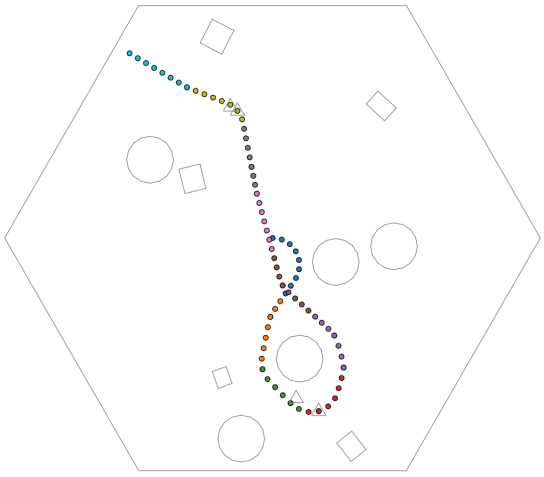
(a) Combinatorial, Time limit = 120s



(b) Combinatorial, Time limit = 30s

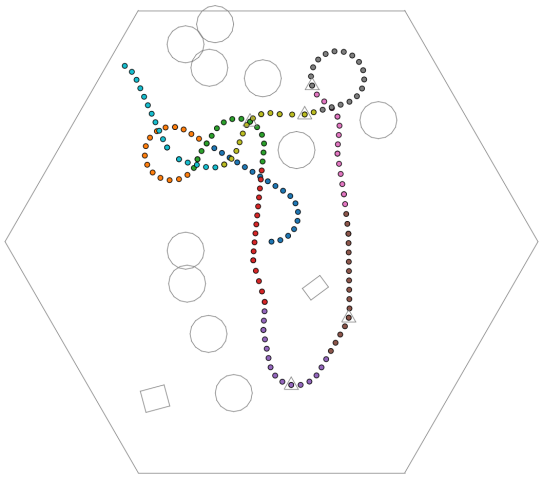


(c) Sampling, Time limit = 120s

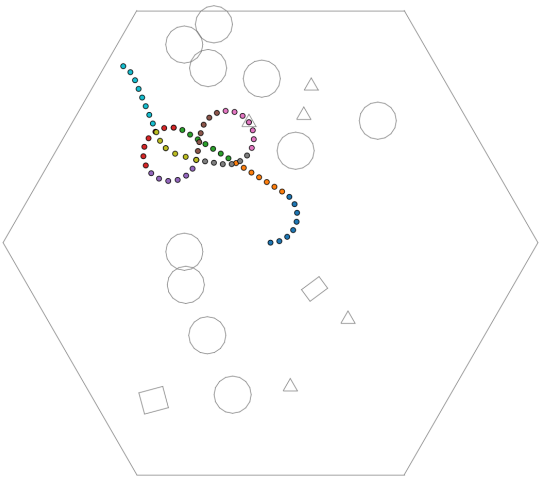


(d) Sampling, Time limit = 30s

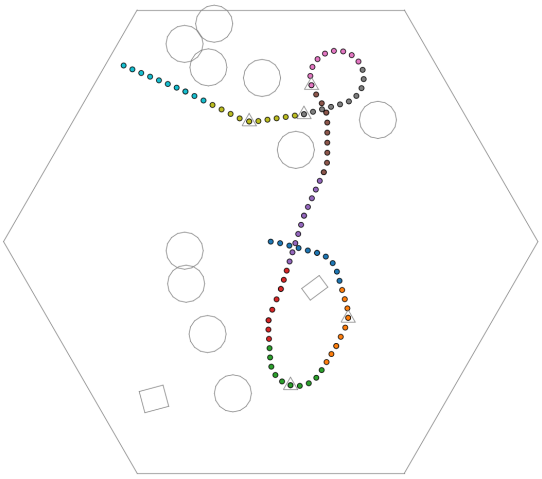
Figure 6: scenario B



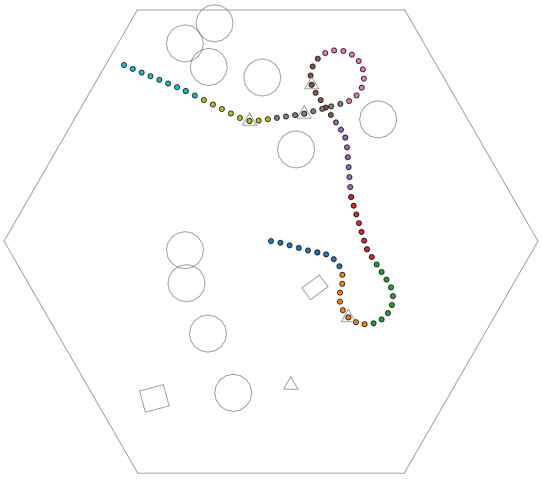
(a) Combinatorial, Time limit = 120s



(b) Combinatorial, Time limit = 30s



(c) Sampling, Time limit = 120s



(d) Sampling, Time limit = 30s

Figure 7: scenario C