CS517 Final Report: Minimum Obstacle Removal

Andrey Kornilovich; Chunxue Xu

June 9th, 2021

1 Introduction

In the robotic path planning field, navigation of an environment is a core challenge. There are many real world examples of environments of that contain obstacles of varying risk that must be crossed to go from a start point to a target point.

The *minimum constraint removal problem* was introduced by Hauser (2012). The objective of this specific motion planning problem is to remove the fewest geometric constraints (i.e., obstacles) necessary on a path to connect a start and target state. The optimal solution could be potentially found after the subsets of obstacle set are enumerated. Hauser introduces Continuous and Discrete minimum constraint removal problem, and proved that the decision version of this problem (i.e., is there a set of n constraints such that it is reachable from the start state to target state via this set) is NP-complete by a polynomial-time reduction from minimum set cover problem.

Erickson et al (2020) discussed a specific situation of the *minimum constraint removal problem*, in which the obstacles are restricted to convex polygonal sets. They proposes an environment definition for the robot to navigate through to formulate the scenario as an NP-hard problem. The robot will be represented by a object in 2d space, along with a set of obstacles, which are represented as lines in 2d space.

In this report, we studied a simple case of minimum constraint removal problem on a 2d grid space (dimensions n*n on R^2), along with a set of M obstacles $\{O_1, O_2, ..., O_m\}$, which are represented as blocked nodes in the 2d grid space. The obstacles are restricted as polygons (i.e., rectangles) with a cost weight. For this exploration of the *minimum constraint removal problem*, there exists a robot start point q_s and an target point q_t located at the grid map corners, with y being a predetermined path through the graph between these two points. The objective of this study is to find the constraint removal with minimum cost for a given path, by encoding an obstacle map representation and robot path as a Boolean SAT formula, and finding the optimal solution by checking the decision problem of removing a set of n constraints to construct a collision free path using a SAT solver.

This type of graph theory problem and decision making has many practical applications in the real world, such as robot navigation and path planning as mentioned earlier, but also other areas such as congestion/bandwidth estimation for road navigation and internet traffic.

A set of command-line tools written in Python are included alongside this report in order to generate obstacle maps, and construct and solve Boolean formulas, whose literals represent the set of obstacles to remove to clear the chosen path with the smallest overall cost for a given map.

2 Data and Methods

2.1 Map Data

The input environment for robot navigation is represented as 2D grid maps. We test 2D maps with a set of parameters, including the map size (e.g., 5 * 5; 100 * 100; 500 * 500), the number of obstacles (e.g., 5, 10, 15) and the cost weight to remove an obstacle. The obstacles are represented as occupying the nodes. The robots can go through free nodes from the initial location (e.g. point q_s) to its end location (e.g. point q_t). For the simulation purpose, we constructed the a set of 2D grid maps as input cases by randomly setting obstacles within the grip map. By encoding this problem to the SAT solver, we evaluated how long it takes to solve the input instances of different map sizes and obstacles to get the boundary between feasible and infeasible input sizes.

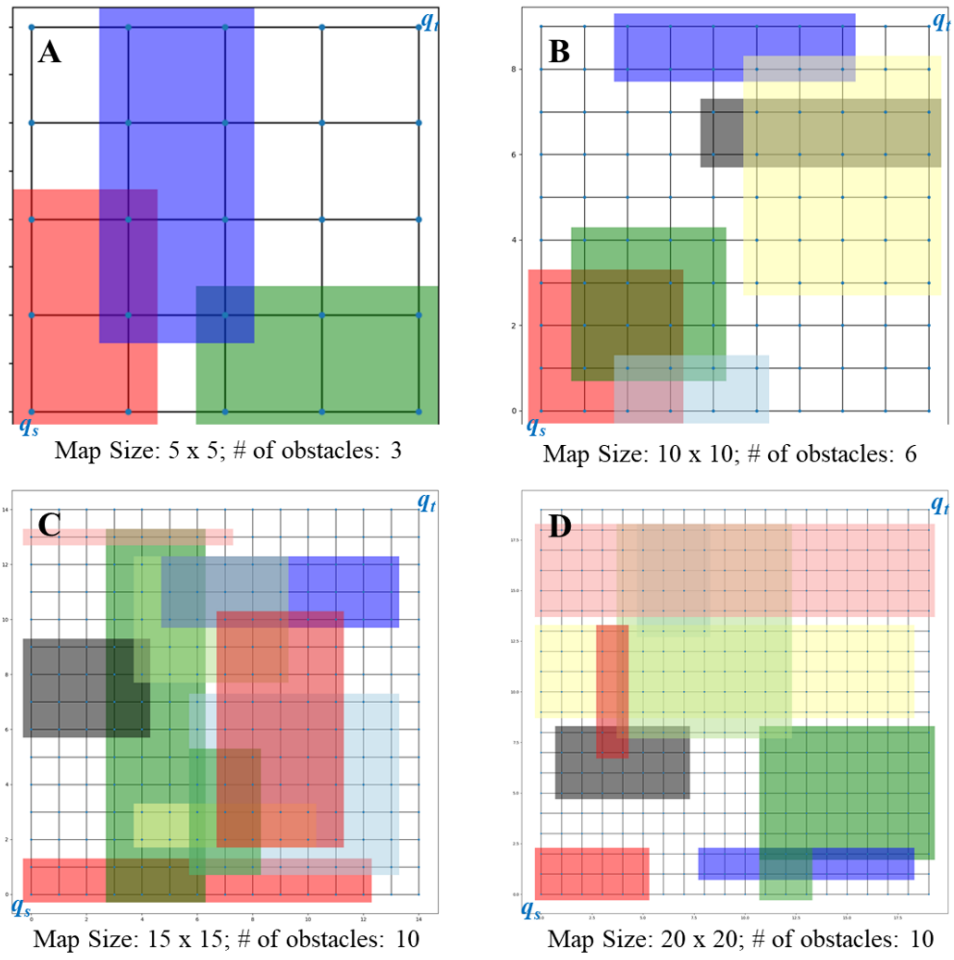

Figure 1: Examples of grid maps with some random rectangle obstacles blocking the path from q_s to q_t. At least a set of two obstacles has to be removed for Map (A), (B) and (C); and a minimum set of three obstacles should be removed for Map (D). A random cost weight (See figure 2) was set for each obstacle so that a optimal road path with minimum cost can be tested.

	x_min	x_max	y_min	y_max	weights
Map dimension	5				
Number of obstacles	3				
Obstacle #1	0	1	0	2	1
Obstacle #2	2	4	0	1	2
Obstacle #3	1	2	1	4	2

Figure 2: Map table example showing the parameter settings for Map (A).

2.2 SAT encoding, obstacles, and gadgets

To encode the above graphs as a Boolean expression, we first decide on one of 3 predetermined s-t path types to construct based on the n x n graph. We have the option of "diagonal", "up-right", and "right-up", stored as a set of (x,y) coordinates that the robot must follow. These paths all end up being the same length, of $k = (2n) - 1$. To construct a Boolean formula to solve this path, we will construct k clauses, where each clause represents a point on the robot's path. In order for the entire formula to be satisfiable, each coordinate in the robot's path much be satisfied. The criteria for a clause/point to be satisfied, is that if the point is covered by any amount of obstacles, at least one obstacle must be removed. The obstacles are our literals, where True means the obstacle is removed, and False means it is not removed. If the point is already clear of obstacles, it is trivially satisfied. Needing to remove at least 1 obstacle per clause was chosen as the criteria instead of removing all, because if all obstacles needed to be removed in the path's way, it would not be an NP-hard problem anymore due to the fact that the paths are predetermined (simply remove all intersecting obstacles). As a real world analogous, for the robot to cross this point on the graph, it must chose at least one obstacle to traverse over. Once it traverses the chosen obstacle, it does not need to cross any other obstacles to clear that same point, and moves on to its next coordinate.

$obstacles = \{O_1, O_2, ..., O_m\}$

$path = \{(x_1, y_1), (x_2, y_2), ..., (x_k, y_k)\}$

$intersection = \{o_i \text{ if } p \in obstacles, \forall o \in obstacles, \forall p \in path\}$

$formula = \{pt\_clause(p_1) \wedge pt\_clause(p_2) \wedge ... \forall p \in path\}$

$pt\_clause(p) = \{intersection(o_1) \vee intersection(o_2) \vee ... \forall o \in intersection(p)\}$

At this point, we have constructed the Boolean formula to pass into our chosen SMT solver. By passing the formula into the solver, we get back an assignment of obstacles that satisfies the formula. However, it is clear that most graphs will have multiple solutions that will satisfy the formula, and we would like to be able to find all possible solutions to calculate the one with the smallest cost. To accomplish this, after finding a solution, we add the negation of that result as another clause to the Boolean formula before re-running the solver. This forces the SAT solver to not output the same result again, and this can be done until all possible solutions are found.

$results = map\_sat\_solver(formula)$

$formula = \neg(results) \wedge formula$

This is the pseudo-code that the solver follows to find all solutions (see *map_sat.py*) for a given path based on the above formulation:

```
solve(points, obst):
    solutions = []
    formula = And(Or(obst[i] if p covered by obst[i]; else True) for i in obst
                  for p in points )

    while(True):
        result = solve(formula)
        if(result == SAT):
            solutions.append(result)
            formula = And(!result, formula)
        else UNSAT:
            break
```

3 Implementation and Example

The included scripts go through all the steps required to generate a map, building the Boolean formula, and finding all solutions and costs. In this section let's use the scripts to find the best path through a new obstacle map. Start by generating a new obstacle map of size 5x5, with 5 random obstacles.

```
python3 utils.py --generate 5 5
```

Graph the obstacle map:

```
python3 utils.py --graph mapData/5x5_5blocks.csv
```

Obstacle map input csv and graph png files can be found in the *mapData* directory. Here is the csv output of our 5x5 5 block test (see figure 2 for formatting of csv file):

map				
5				
5				
0	2	0	2	3
3	3	2	4	1
1	3	3	3	4
0	2	2	3	1
1	3	0	0	3

Table 1: mapData/5x5_5block.csv

Run the obstacle map csv through the SAT solver and automatically graph (can also graph the .csv results separately later):

```
python3 map_sat.py mapData/5x5_5blocks.csv --graph
```

Let's look at the results for all 3 paths. Result data and graphs are in the *resultData/* directory, and the intermediary solver steps are logged to *info.log*. Here are the solution csv files:

diagonal				
5				
2				
3	3	2	4	1
0	2	0	2	3

right				
5				
2				
1	3	0	0	3
0	2	0	2	3

up				
5				
3				
3	3	2	4	1
0	2	2	3	1
0	2	0	2	3

Table 2: mapData/5x5_5block_diagonal.csv & ..._right.csv & ..._up.csv

Now let's look at *info.logs* to see the steps the script took to solve the input instance for each path. The full log file for this specific test case is stored as *example_info.log* in the code repository for reference. The logs will first contain info about the csv data it just read in (Table 1). After that, we get printouts of the constructed Boolean formula's for the different path types. For this example let's focus on the results of the diagonal formula, as it has multiple obstacle removal solutions, unlike the up-right and right-up solutions which only have possible 1 assignment that satisfies their formulas.

```
...

Find all SAT solutions for diagonal path
----------------------------------------
Boolean formula of path:
full      : (o0 & (o0 | o4) & o0 & o0 & (o0 | o3) & o1 & (o1 | o2) & True & True)
simplified: (o0 & o1 & (o0 | o4) & (o0 | o3) & (o1 | o2))
obstacles : frozenset({o0, o1, o2, o3, o4})

SAT solution:
o2 := False
o1 := True
o3 := False
o0 := True
o4 := False

SAT solution:
o2 := False
o1 := True
o3 := False
o0 := True
o4 := True

SAT solution:
o2 := False
o1 := True
o3 := True
o0 := True
o4 := True

...

SAT solution:
```

```
o2 := True
o1 := True
o3 := False
o0 := True
o4 := False

UNSAT, no more solutions.

...
```

This snapshot does not include all solutions to the diagonal path, just the beginning and end of the printout (8 solutions were found). Using the gadget for adding previous results back into the Boolean formula, the solver is able to loop until the formula is UNSAT, giving us all the possible solutions. The Boolean formula, both full and simplified, as well as the obstacles that intersect the chosen path are logged as well at the beginning of the process. Up-right and right-up paths are logged after the diagonal path.

After all the solutions for each path are found, the end of the log file shows the cost associated with each solution found, as well as the final results of the best solution and cost for each path type. Graphs of the input case and solutions are on the following page.

```
...
Calculate diagonal path solution costs
--------------------------------------
cost of solution 1: 4
cost of solution 2: 7
cost of solution 3: 8
cost of solution 4: 5
cost of solution 5: 9
cost of solution 6: 12
cost of solution 7: 11
cost of solution 8: 8
minimum cost found: 4

Calculate up-right path solution costs
--------------------------------------
cost of solution 1: 5
minimum cost found: 5

Calculate right-up path solution costs
--------------------------------------
cost of solution 1: 6
minimum cost found: 6


Final results
------------------------------------------------------------
cost of best diagonal path: 4
cost of best up-right path: 5
cost of best right-up path: 6
```

```
best diagonal solution:
o2 := False
o1 := True
o3 := False
o0 := True
o4 := False

best up-right solution:
o1 := True
o3 := True
o0 := True

best right-up solution:
o4 := True
o0 := True
```

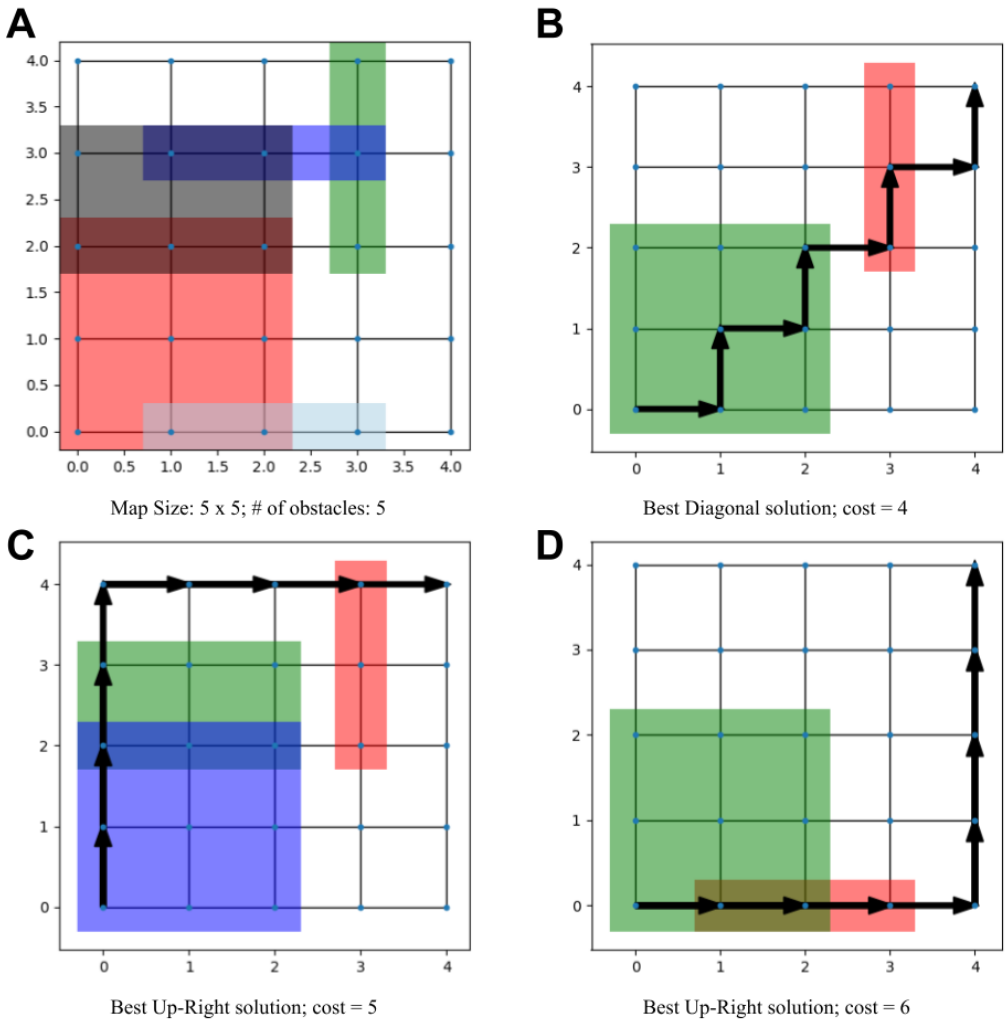

Figure 3: (A) is the obstacle map. (B), (C), and (D) are the best solutions for each path type.

The scripts were written in python, leveraging the pySMT API in order to easily interface with the MathSAT 5 SMT solver for constructing and solving our Boolean formula, and parsing the results. Standard python libraries were used throughout for file I/O, argument parsing, and logging. NetworkX and Matplotlib were used to generate the plots. See the included source code or the github repo (3) for more details, and to see the exact input, output, and logging data of the example that was covered in this section.

4 Evaluation

We evaluated the solver's performance on 2D grid maps of different sizes and with different number of random rectangle obstacles. The running time was tested for two cases:

1. The change of the running time with increasing the map size from 10*10 to 500 * 500, when the numbers of obstacles were fixed as 5, 10, 15 and 20;
2. The change of the running time with increasing numbers of obstacles from 1, 2,.. to maximum, when the map sizes were fixed as 10*10, 50*50, 100*100, and 500*500.

The experiments were performed using a workstation with a Intel® Core™ i7-6800K Processor (16 GB RAM, up to 3.40 GHZ). Considering the randomness in generating the obstacle (i.e., the random size and location on the 2D grip map), each testing case of a set of obstacle parameters were repeated 6 times to get the mean value of the running time. The results are shown in Figures 4-5.

With the map size fixed, running times are roughly increasing exponentially with increasing the number of obstacles. As is see in figure 4, the behaviors of the solver were similar with increasing the number of blocks on a set of maps of varying sizes. In particular, with more than 10 obstacles, it took more time to get the solution on a smaller map (e.g., about 120s for 10*10 maps with 15 obstacles). So the obstacle distribution (e.g., sparse or overlapping) can potentially affects the solver performance dramatically.

Compared to the number of blocks, the map size has less impacts on the solver's running time. As is seen in figure 5, the running time is roughly stable on the testing range of varying map sizes with number of block fixed. The exception case is that it took longer time to get the solution than averaged cases on a smaller map with relatively more obstacles, just as what is shown in figure 4.

The solver scales easily with the map size, but poorly with the number of obstacles. The instances were unsolvable (do not terminate or running time is more than 1h) when there were more than 20 obstacles. A set of 2D grid maps (of which the sizes varying from 10*10, 100*100, 500*500, to 1000*1000) with 20 obstacles were tested and this boundary between feasible and infeasible input cases was validated.

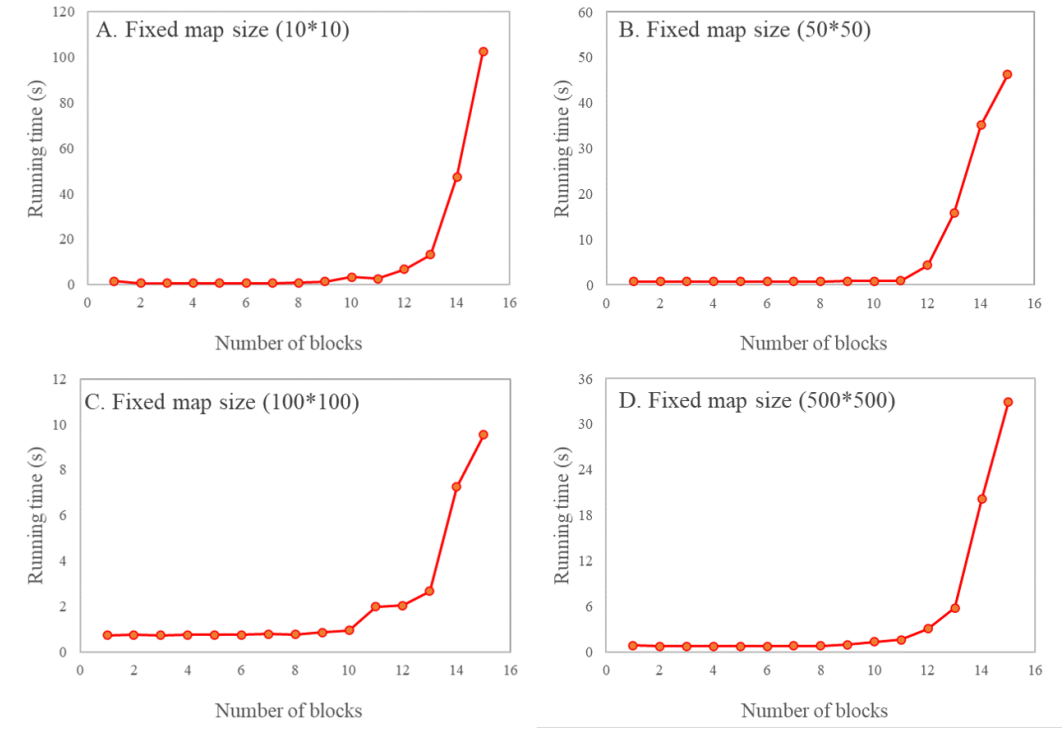

Figure 4: Running time analysis for maps with fixed sizes and varying numbers of blocks. The averaged time were obtained with six repeated runs on maps with random blocks.

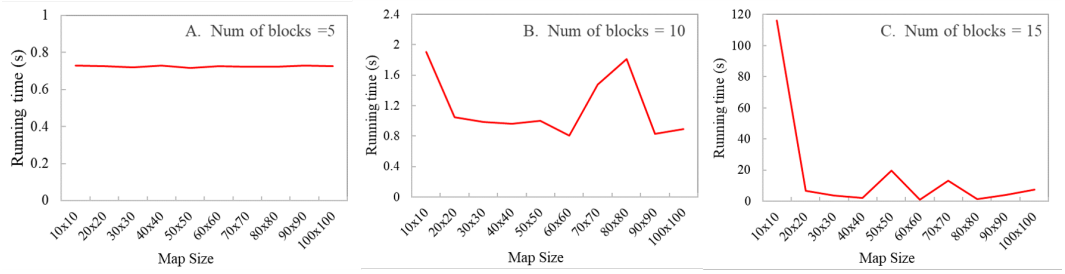

Figure 5: Running time analysis for maps varying sizes and fixed number of blocks. The averaged time were obtained with six repeated runs on maps with random blocks.

5 Conclusions and future work

In this study, we demonstrate a solver for *minimum constraint removal problem* in the robotic path planning field, with a specific focus on the 2D grid environment and rectangle obstacles blocking the three optional paths (i.e. diagonal, up-right and right-up) from start node to the target node. By reducing the *minimum constraint removal problem* to the SAT problem, the path planning instances were encoded in SAT and can be solved using a SAT solver.

There are many different adaptations that can be potentially tested in the future:

1. We tested three optional predetermined paths on 2D grid map as a navigation environment. It would be interesting to extend the solver to a path planning problem with free navigation on the 2D map. How could the solver help inform us if a longer more windy path leads to a smaller overall cost?

2. We assigned random weights to the obstacles to represent the cost for obstacle removal. More complicated situations can be tested under a real-world scenario. For example, the cost can be related to some settings such as obstacle size or accessibility from the start node.

3. Rectangle obstacles were randomly generated on the map, which can be potentially extended to convex polygon obstacles.

4. The performance of the solver can be compared with other state of art algorithms that supplement their decision making with a greedy heuristic before further optimization.

References

[1] Erickson, L., & LaValle, S. (2013). A Simple, but NP-Hard, Motion Planning Problem. Proceedings of the AAAI Conference on Artificial Intelligence, 27(1). Retrieved from https://ojs.aaai.org/index.php/AAAI/article/view/8545

[2] Hauser, K. 2012. The minimum constraint removal problem with three robotics applications. In Proc. Workshop on the Algorithmic Foundations of Robotics.

[3] Andrey, K & Chuxue X. (2021). Application of minimum constrain removal of a path on a 2D grid. https://github.com/Andrey-Korn/CS517-final