# Robust Plan Inference in the Keys and Doors Problem

## Creating Robust Plans using Replanning

**Koen van der Knaap**[1]
**Supervisors: Sebastijan Dumančić**[1]**, Issa Hanou**[1]**,**
**Reuben Gardos Reid**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Koen van der Knaap
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Issa Hanou, Reuben Gardos Reid, Merve Gürel

**Abstract**

Planning is very important in everyday life, whether it would be creating schedules for planes or plans for manufacturing. These domains contain uncertainties requiring plans that are robust. However, there is a need for an approach which creates robust plans regardless of the domain and without changing its planning agent. Here, a replanning approach is proposed akin to the Metropolis-Hastings algorithm and its performance is compared to the performance of importance sampling. Replanning works by iteratively trying to improve the previously generated plan. The performance is compared by means of the Keys and Doors problem. It is found that replanning performed better than importance sampling in the two analysed problems. Furthermore, changing the parameter, $\sigma$, used in the replanning approach showed a significant difference in its corresponding performance. While the replanning approach has only been tested on the Keys and Doors problem, the results show that replanning is a promising approach which could work irrespective of the domain at hand.

# 1 Introduction

Planning is something which is very important in day-to-day life, whether it would be creating schedules for planes in airports or manufacturing. Unfortunately, the problems that need to be tackled each day often involve a lot of uncertainties. In the case of manufacturing, some processes could take longer than expected and possibly even fail. A plan consists of a sequence of actions that aim to achieve an end goal. Naturally, one would re-plan if an issue arises during plan execution in order to fix the errors which have just occurred. However, when there is no feedback after executing a step, because e.g. a sensor broke, we would not be able to use replanning. That is why the plans need to be made *robust*. Robustness in a plan will give us a lower chance of failure, even when there are conditions that deviate from what is expected.

The methodology in this paper has been inspired by Gardos Reid [1]. In his work, a rail network simulator was used in combination with a probabilistic programming language to infer robust plans for the train unit shunting problem. Probabilistic programming allows the programmer to express a model probabilistically [2]. The problem can be formalised using the probabilistic model and the model can be solved using a planner without modifications. The methods Gardos Reid used to infer robust plans were importance sampling and Metropolis-Hastings [3]. In his Metropolis-Hastings approach, a new plan is proposed based on where the previous plan failed. In the train unit shunting problem, it is possible to know if a plan has failed, i.e. a delay has occurred, before executing the entirety of a plan. In other domains it may not be possible to know if a plan failed at an intermediate step, making it hard to generalize this method.

The question this research aims to answer is whether the Metropolis-Hastings approach can be modified to be applicable in general domains. To be more specific, this research aims to answer whether a modified replanning approach could be used to create robust plans for the Keys and Doors problem, which is used as a case study. First, the uncertainties that might exist in the real-world version of the Keys and Doors problem need to be explored. Furthermore, this research investigates the performance difference between the modified replanning approach and importance sampling. Lastly, it also aims to answer what parameter settings achieve the best performance in replanning.

This research aims to solve these questions by using a Probabilistic Programming Language (PPL) to create robust plans for the probabilistic version of the Keys and Doors

problem provided by [4]. The Keys and Doors problem serves as an indicator for the performance of the inference techniques. In the PPL, a probabilistic model of the Keys and Doors problem will first be created. Two different inference techniques called importance sampling and replanning, which is inspired by Metropolis-Hastings approach proposed by Gardos Reid in [1], will then be used to infer robust plans. Importance sampling and replanning are subsequently compared based on the robustness of their proposed plans in two example scenarios of the Keys and Doors problem.

Showing that robust plans can be made without looking into the current state of the problem, could create new possibilities for domains where robust planning is critical, especially in domains in which it is either hard or even impossible to sense the environment. In the future, robust plans could enable planning in domains where sensing is too demanding or replanning is not possible during plan execution. For example, chemical plants could operate on a single plan without having to watch temperatures, pressures, volumes, and still be able to synthesize chemicals.

The paper is structured as followed. Section 2 outlines previous uses of probabilistic programming. In Section 3, the Keys and Doors problem is defined and its uncertainties are explored. Furthermore, the inference techniques are described in Section 4. The results of the experiments are highlighted in Section 5. Next, the reproducibility of the results and ethical implications of this research are laid out in Section 6. In Section 7, the results are analysed and discussed. The research questions are answered in Section 8. Moreover, directions for future research are also explored.

# 2 Background

Probabilistic programming makes it easier for the programmer to define probabilistic models and reason about the models [2]. Before probabilistic programming, creating a probabilistic model would require manually computing the likelihoods and other relevant metrics of the model. The probabilistic programming language helps the programmer by automatically calculating the relevant statistical metrics. Moreover, changing a probabilistic model is less cumbersome as only the model parameters and the associated code, calculating the metrics, have to be altered. Furthermore, these probabilistic programming languages often embed inference algorithms, such as importance sampling and Markov Chain Monte Carlo, within themselves allowing people to apply these inference algorithms more conveniently.

Probabilistic programming has already been applied to facilitate probabilistic inference in various domains. Gardos Reid used probabilistic programming to infer robust plans for the train unit shunting problem (TUSP) [1]. In his master's thesis, he applied the Metropolis-Hastings algorithm to TUSP. In each iteration of the Metropolis-Hastings approach, a new plan would be proposed based on where the previous plan failed. A major advantage of this method is that it does not require modifying the planning algorithm or simulator. It is therefore able to take advantage of previous research on the TUSP. However, his approach is not able to be generalized to other domains, because it may be impossible to determine at what step a plan failed in other domains. Furthermore, in [5] by Semenova et al. a probabilistic programming language has also been used to create a Bayesian Neural Network which predicts the toxicity of chemical compounds. Moreover, Deng et al. introduce a system to identify different driving styles [6]. To this end, a probabilistic model was used to better categorize driving styles using probabilistic models. These examples highlight that probabilistic programming is already prevalent in numerous domains.

A probabilistic model in a probabilistic programming language can be used to infer posterior distributions given observations. An observation is a measurement of the realization of a random variable. Monte Carlo methods can be used to perform probabilistic inference. Importance sampling is one of the most basic Monte Carlo methods. It works by sampling from a non-target distribution and scaling the weight of the samples based on the target distribution. Another Monte Carlo method is Markov Chain Monte Carlo (MCMC). A special case of MCMC is the Metropolis-Hastings (MH) algorithm introduced by Hastings in [3]. The main idea of MH is to propose new samples based on the previous sample. The new sample is either accepted or rejected based on the weight of the newly proposed sample and the weight of the previous instance.

# 3 Methods

This section outlines the Keys and Doors problem and the possible uncertainties therein. First, a detailed description of the original Keys and Doors problem, without uncertainties, is given. Using the description, different possible uncertainties are provided and analysed.

## 3.1 The Keys and Doors Problem

The Keys and Doors problem has been taken from the `pddlgym` library[1] and slightly modified such that uncertainties have a more substantial effect [4]. In the original problem, a player needs to reach a square by picking up keys in order to unlock rooms to reach its final destination. An example instance of the Keys and Doors problem can be found in Figure 1.

The Keys and Doors problem is formally defined using a Planning Domain Definition Language (PDDL). A PDDL allows us to define different domains formally in a human readable format [7]. PDDL was first introduced in 1998 by Ghallab et al. in [8]. It separates the definition of the domain and problem. The domain contains predicates (logical facts) and actions. Actions have preconditions, which must be met before the actions can be applied. Actions also have effects, which change the current predicates.

The Keys and Doors problem is similar to a normal pathfinding problem, where a player needs to reach a goal position. In contrast to a normal pathfinding problem, there exist rooms which the player is not able to enter unless it picks up the corresponding key which unlocks the locked room.

The player is able to modify the scenario by executing an action. Here, the problem differs from the definition given in the `pddlgym` library. In the `pddlgym` library, the player is able to teleport from one square to another, i.e. the squares do not have to be adjacent. However, to make the problem more realistic, in our version of the problem, the player is only able to move to adjacent squares. All the possible moves the player can make are:

- $l$, Moving left
- $r$, Moving right
- $u$, Moving up
- $d$, Moving down
- $p$, Picking up a key at the current position

[1] https://github.com/tomsilver/pddlgym

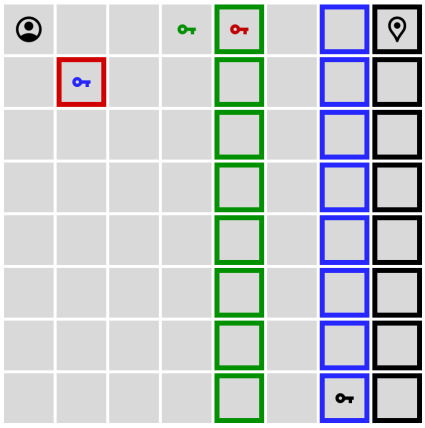


Figure 1: Example of the Keys and Doors problem containing four rooms and corresponding keys. Squares with the same colour as a key, are opened by picking up that key. The starting location is located in the top left. The goal location is located in the top right.

The player is able to achieve the goal position in a scenario by executing a plan. A plan consists of a finite number of actions. Each of the actions in the plan will be applied to the problem sequentially. If an action in the plan is not able to be executed, the state of the scenario will remain the same, i.e. the action will not be executed and no error will be raised. A plan is defined to reach the goal if, all of the actions in this plan have been executed sequentially and the player is at the goal position afterwards. The plans are created using Fast Downward[2], a planner for PDDL problems. An example solution to the problem in Figure 1 would be:

$[r, r, r, p, r, p, d, l, l, l, p, r, r, r, r, r, d, d, d, d, d, d, p, u, u, u, u, u, u, u, r]$

## 3.2 Uncertainties

In order to test the effectiveness of inference techniques, the Keys and Doors problem needs to contain uncertainties. It would be too simple to generate robust plans otherwise, as using the plan for the original problem would always be robust. The problems including their uncertainties are modelled using a probabilistic programming library for Julia called `Gen`[3], developed by Cusumano-Towner et al. [9]. `Gen` allows us to write models and use them to implement probabilistic inference more easily.

Adding uncertain starting positions for the player, makes the player unsure what its current position is in the scenario. The player is able to alleviate the uncertainties by making moves which are not allowed. One example of this would be moving into a wall. If the player were to either be right next to the wall, or one space away from the wall, the player could make a move towards the wall. After executing the move, the player is at the exact same position regardless of starting position.

Unlocking a portion of the initially locked rooms, could help the player create plans which are shorter. However, the shorter plans created may not be robust, as the room may still be locked. When giving a chance larger than 50% to rooms being unlocked, the player should

[2]https://www.fast-downward.org/latest/
[3]https://www.gen.dev/

skip picking up keys if it would use the most likely scenario to create its plan. Unfortunately, we cannot lock rooms which were unlocked in the original version of a scenario since this would likely lead to the modified scenario becoming impossible.

Making the location of the keys uncertain, would force the player to try to pick up the keys at multiple locations. This does, however, increase the length of the plans. The player could also choose to not pick up the key in its entirety by devising a plan which does not move into the corresponding room. Still, in some problems, the player may be required to move into a room whose key's position may be uncertain. In that case, the player should always prefer picking up the key.

An uncertain goal position is an uncertainty which does not help to discover the efficiency of inference techniques. An uncertain goal position makes it impossible to create robust plans in certain scenarios. As mentioned in Section 3.1, the player must stand on a goal after all of the moves have been executed. Since it is impossible for the player to know which square it needs to reach, it is impossible for a plan to exist which would work in a relatively high proportion of the uncertain scenarios. In other words, an uncertain goal position would make the problem too difficult.

# 4 Inference Techniques

Given the probabilistic model for the Keys and Doors problem, we can start to use inference techniques to infer robust plans. The used methods are importance sampling and replanning. The goal of these methods is to create plans which are robust. The robustness of a plan is defined as the percentage of uncertain problems the plan is able to solve.

## 4.1 Importance Sampling

Importance sampling creates plans for random realizations of the original problem. `Gen` has a method to implement importance sampling more easily, named `importance_sampling`[4]. This method accepts a generative model, and runs this model $n$ times in a loop. It subsequently outputs the traces, their weights, and the marginal likelihood of the observations. Before actually running importance sampling, $m$ random problems are first sampled. The random problems allow us to estimate the robustness of plans.

The generative model used in importance sampling contains three core steps. First, a random problem is sampled from the original problem. Second, a plan is created for the randomly sampled problem. Lastly, an estimate for the robustness of the plan is given by running the created plan on the collection of $m$ randomly sampled problems. The code used to run importance sampling can be found in Listing 1.

[4]https://www.gen.dev/docs/stable/api/inference/importance/#Gen.importance_sampling

```
@gen function solve_kad(domain, problem, generated_problems, planner)
    # Generate random plan
    random_plan ~ create_random_plan(domain, problem, planner)
    # Rank the plan
    plan_rank = rank_plan(random_plan, generated_problems)

    # Generate bernoulli random variable
    {:robustness} ~ Gen.normal(plan_rank.robustness, 0.05)
    return plan_rank
end

generated_problems = [create_random_problem(problem) for _ in 1:m]
traces, log_norm_weights, lml_est = Gen.importance_sampling(
    solve_kad,
    (domain, problem, generated_problems, Planner.planner),
    choicemap(:robustness => 1),
    n
)
```

Listing 1: Generative function and its use in importance sampling

## 4.2 Replanning

Replanning tries to improve the plans slightly in each iteration of the Metropolis-Hastings loop. It is inspired by the Metropolis-Hastings approach proposed in [1] by Gardos Reid and sought to be generalized to all possible problem domains. Moreover, it uses a modified version of `Gen`'s Metropolis-Hastings function[5]. The `metropolis_hastings` method normally executes one step of the Metropolis-Hastings algorithm loop, returns a trace and returns whether the newly generated trace was accepted.

Similar to importance sampling, $m$ random problems are first created to estimate the robustness of the plans. In addition to the random problems, an initial trace is generated, which serves as a basis to generate new plans from. The Metropolis-Hastings loop is now entered, in which we will generate new plans. This loop is executed $n$ times, generating $n$ plans. There are three main parts in each step of the loop. The code used for proposing a new plan is located in Listing 2. First, a random problem is generated. We then check whether the previous plan would have reached the goal in this random problem. If it does not, a random prefix of the previous plan is selected and a plan is appended in order to succeed in the randomly generated plan. Next, we rank the robustness of this new plan. A weight is assigned to this plan based on a normal distribution. The weight is equal to the probability density of a normal distribution at 1.0. The normal distribution is defined with the mean equal to the robustness of the proposed plan and standard deviation $\sigma$, named `sd` in Listing 2. $\sigma$ can be modified to alter the behaviour of replanning.

[5] https://www.gen.dev/docs/stable/api/inference/mcmc/#Gen.metropolis_hastings

```
@gen function propose_new_plan(
    prev_trace, problem, problems, sd, domain, planner
)
    prev_plan = prev_trace.retval.plan
    random_problem ~ create_random_problem(problem)
    new_plan = prev_plan
    if !execute_plan(random_problem, prev_plan)
        # Random prefix from the previous plan
        r_index ~ Gen.uniform_discrete(1, length(prev_plan))
        partial_plan = prev_plan[1:r_index]

        curr_state = execute_plan_state(random_problem, partial_plan)
        # Plan from that point onward
        additional_plan = planner(domain, curr_state_python)
        new_plan = vcat(partial_plan, additional_plan)
    end

    plan_rank = rank_plan(new_plan, problems)
    {:robustness} ~ Gen.normal(plan_rank.robustness, sd)
    return plan_rank
end
```

Listing 2: Generative function used to propose new plans. First a random problem is generated, and a new plan is proposed for this problem if the goal is not reached using the old plan.

# 5 Results

This section displays the results of running importance sampling and replanning. The experimental setup is explained first, where the parameters used for the experiments are laid out. Secondly, the results of running importance sampling and replanning on two vastly different problem instances are described.

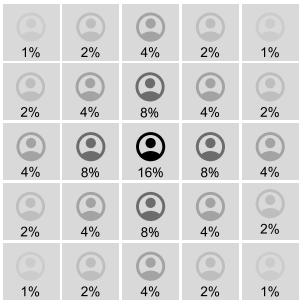
## 5.1 Experimental Setup

The experiments were run on a computer with a Ryzen 5 5600X processor[6] and 16 gigabytes of RAM. No specific timeout has been set for the experiments. The inference techniques have been run on one custom problem and one problem taken from the `pddlgym` library[1]. Plans were created using the Fast-Downward planner located in the accompanying `pddlgym_planners` library[7].

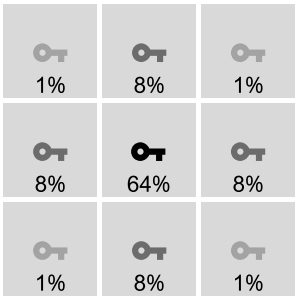
The location of the player, the location of the keys, and whether the rooms are unlocked were made uncertain by modifying the problem. The player's starting location was modified both in horizontal and vertical offsets. In both directions, the chance of staying on the same block is 0.4. The probability of moving in the positive or negative direction is 0.2 if moving one block, and 0.1 if moving two blocks. The location of the keys was modified

[6] https://www.amd.com/en/products/processors/desktops/ryzen/5000-series/amd-ryzen-5-5600x.html
[7] https://github.com/ronuchit/pddlgym_planners

in an analogous way but with different probabilities. There is a probability of 0.8 to stay on the same block and a chance of 0.1 to move one block in either the positive or negative direction. The keys must stay in their original room. If a certain target square was infeasible, its probability was set to 0.0 and redistributed among the other remaining valid options. A previously locked room has a probability of 0.5 of being unlocked in its uncertain version. Figure 2 shows the possible new starting and key locations and their probabilities.




(a) Player location uncertainty




(b) Key location uncertainty

Figure 2: Two of the uncertainties in the Keys and Doors problem and their corresponding probabilities

Both experiments have been run with the same settings. For both importance sampling and replanning, a total of 1000 random problems were generated based on the original problem. These problems are used to estimate the robustness of a proposed plan. Moreover, the total number of iterations in both importance sampling and replanning was set to 1000, meaning that a total of 1000 traces were generated. The $\sigma$'s used in replanning are: 0.2, 0.25, 0.5 and 1.0. These values are chosen because the robustness will always be between 0 and 1 and less robust plans should still be able to be accepted using these values.

## 5.2 Experimental Results

In order to analyse the performance of the importance sampling approach and the replanning approach, we compared the robustness of plans in different problem scenarios. First, their performance was compared in a simple problem, which was a straight line and did not include any keys or rooms. Afterwards, we will visit the example given in Figure 1, which was used in the definition of the Keys and Doors problem.

### 5.2.1 A Problem without Rooms

The goal of this simplified problem is to show why replanning could be a promising strategy. The problem at hand, shown in Figure 3, cannot be solved with a robustness of 100% using importance sampling. This robustness cannot be achieved since there does not exist a random instance of this problem, whose plan will directly work for all random problem instances.

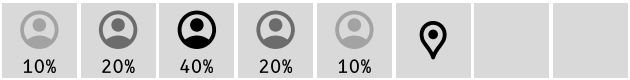
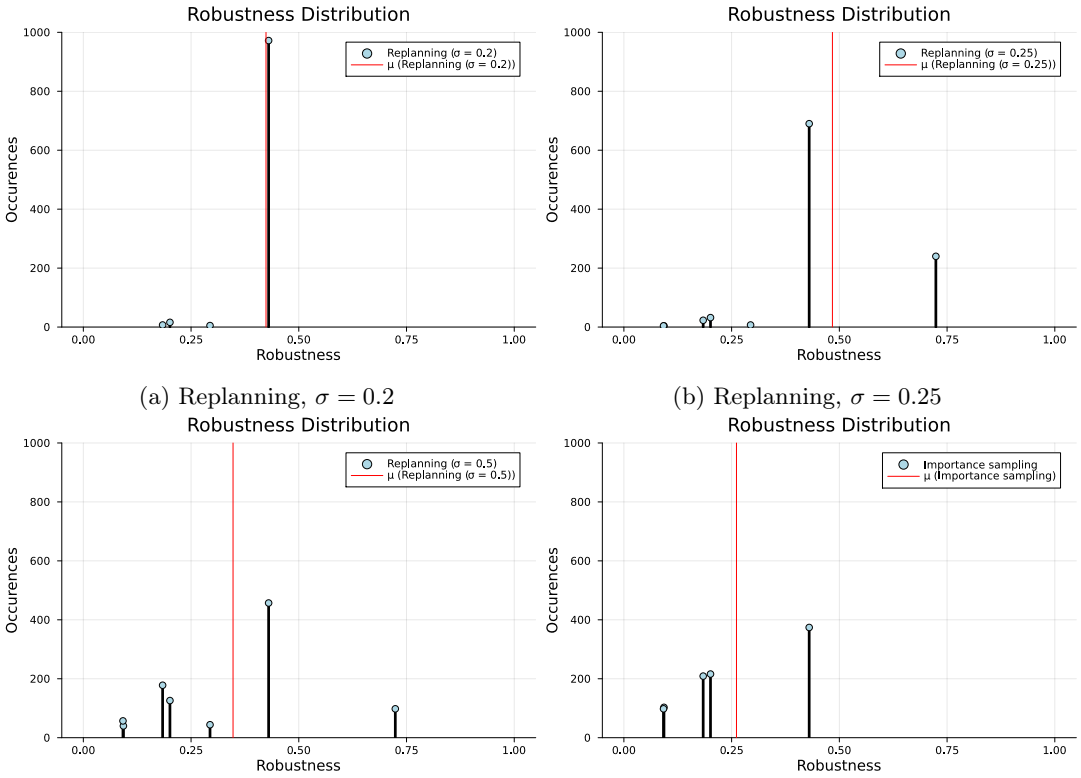



Figure 3: Custom problem forming a straight line without any keys and rooms

Changing the $\sigma$ parameter influenced the results slightly. In Figure 4, we see the number of occurrences of each robustness except $\sigma = 1.0$. The results for $\sigma = 1.0$ are left out for brevity and can be found in Appendix A. $\sigma = 0.25$ produced plans with the overall highest robustness. When $\sigma$ increased, the overall robustness of plans tended to decrease. Especially the number of plans with a robustness $\leq 0.35$ increased when increasing $\sigma$. $\sigma = 0.2$ primarily generated plans with robustness equal to 0.4, but did not generate plans with robustness $\geq 0.5$. In every case replanning performed better than importance sampling.




(a) Replanning, $\sigma = 0.2$

(b) Replanning, $\sigma = 0.25$

(c) Replanning, $\sigma = 0.5$

(d) Importance Sampling

Figure 4: Robustness of plans in the straight-line scenario

### 5.2.2 Revisiting the Example Problem

The problem analysed here is the same as the example problem given in Figure 1. This problem could be considered harder than the straight-line problem analysed in the previous section, since there is a larger number of uncertain scenarios. However, this problem has more possible options where the player might be able to correct itself by trying to execute moves which are not allowed.

The results demonstrate a correlation between the $\sigma$ parameter and the robustness of plans, similar to the previous problem. Figure 5 displays the cumulative distributions of plan robustness. Again, assigning 0.25 to $\sigma$ produced the most robust plans, as the cumulative density function (CDF) shows a sharp increase when the robustness is greater than 0.75. Changing $\sigma$ to 0.2, decreased the robustness of the proposed plans. Similarly, increasing $\sigma$ above 0.25 shifted the CDF left, showing a lowering in the overall robustness of the plans. In comparison, importance sampling performed significantly worse than all of the replanning variants. The plans produced by importance sampling generally had a low robustness value, signified by the sharp increase at low robustness values.

There is a strong correlation between the lengths of proposed plans and the robustness of the proposed plans, as can be seen in Figure 6. The results for replanning with $\sigma = 1.0$ are left out for brevity. Those results can be found in Appendix B. Furthermore, increasing $\sigma$ corresponded to plans of shorter lengths being generated. However, the shorter plans did have a lower robustness than the longer plans.

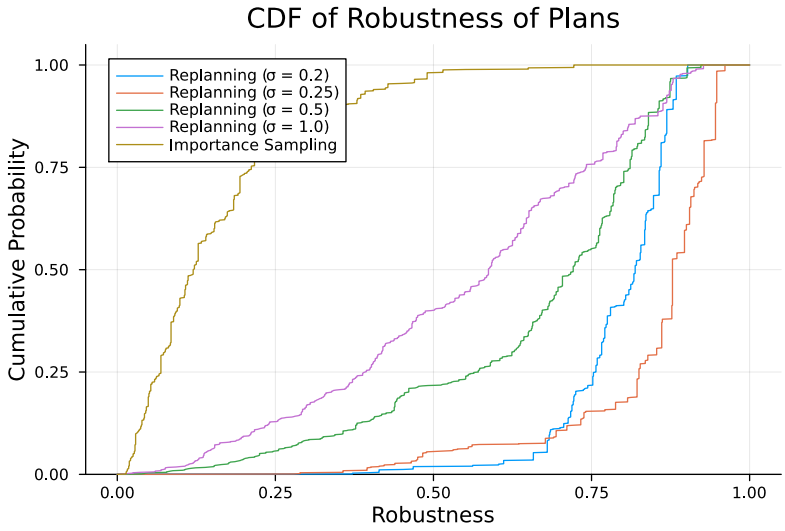
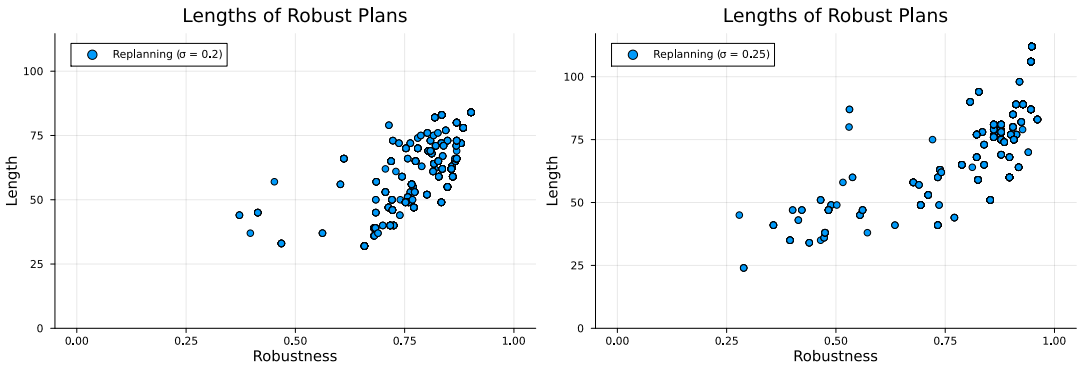


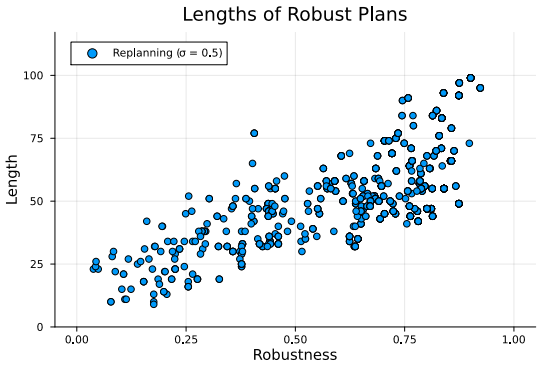

Figure 5: Cumulative distribution of robustness of plans in the example problem



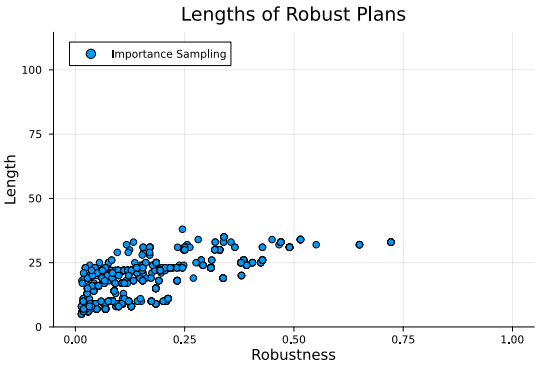

(a) Replanning, $\sigma = 0.2$




(b) Replanning, $\sigma = 0.25$




(c) Replanning, $\sigma = 0.5$




(d) Importance Sampling

Figure 6: Length of proposed plans plotted against the robustness of proposed plans

# 6 Responsible Research

The research conducted during this research project is reproducible within the confines of the Keys and Doors problem. One does need to keep in mind that the results are based on random simulations and may therefore not produce the exact same results each run. Furthermore, due to time constraints and memory limits, only 1000 iterations of both replanning and importance sampling were used. For the same reason only 1000 problems were used to estimate the robustness of the plans, which could create variance in the calculated robustnesses of the plans. A seed is used to make sure that each run of the experiment would produce the exact same results if simulated on another computer. The experiments were run with different seeds as well to make sure that the results were similar across different runs. The code for this project is also available at [10].

No ethical concerns were raised during the research. All of the information used in this research is available online and the data does not include any personal information nor information that could neither harm people nor animals. Because of these reasons there is no risk for privacy violations, nor ethical harm, making this research ethically neutral.

# 7 Discussion

In the experiments, two different scenarios were used to gather results on the robustness of plans generated by importance sampling and replanning. First, these results will be analysed, and an explanation will be given. Afterwards, the limitations of the experiments are laid out.

## 7.1 Findings

Replanning performed better than importance sampling in both the straight-line problem and the example problem. It was especially noticeable in the case of the example problem. In general, the mean robustness of the plans increased as the value of $\sigma$ decreased.

In the straight-line problem, using replanning with $\sigma$ set to 0.2, did not produce plans with a robustness as high as larger $\sigma$'s. One of the causes could be that a less robust plan would need to be accepted before a plan can be generated that has a robustness of 70%. Since $\sigma$ is set to a smaller number, we have a much smaller probability of accepting a plan with a lower robustness. Conversely, the results of the straight-line problem demonstrate that increasing $\sigma$ above 0.2 allows plans with higher robustness to be generated. Adjusting $\sigma$ to a value larger than 0.25 resulted in less robust plans being generated. The less robust plans have a larger probability of being accepted when $\sigma$ is lower. Setting $\sigma$ equal to 0.25 produced the most plans with a 70% robustness. Furthermore, it also produced the highest overall mean robustness. This shows that 0.25 is likely close to the ideal value $\sigma$. Comparing the replanning approach to importance sampling, we see that every instance of replanning outperformed importance sampling.

Replanning also showed improved performance on the larger problem. It created more robust plans when using a lower $\sigma$ as well. However, similar to the smaller scenario, less robust plans were made when $\sigma$ was set to 0.2. The same cause for this could be given as in the smaller problem. In order to reach plans with a higher robustness, we would first need to accept less robust plans. The less robust plans have a significantly lower likelihood to be accepted when $\sigma$ is decreased. However, in the unlikely case that a non-robust plan is accepted and a very robust plan is subsequently generated, the robust plan is almost always retained over succeeding alternatives. Likewise, configuring $\sigma$ to values larger than 0.25 again resulted in plans with lower robustness being generated. Moreover, we similarly observed that $\sigma$ set to 0.25 gave us the most robust plans. Again, we also perceived that every instance of the replanning approach performed better than importance sampling.

The plans made by replanning were significantly longer than the plans made by importance sampling. The plans generated using importance sampling are of optimal length for a random problem instance, but the amalgamation of different plans in replanning makes the plans longer. In general, the more robust a plan is, the longer the plan becomes. There are two reasons for this occurrence. The more robust plans are longer, because the plans need more actions to resolve uncertainties. Additionally, the shorter, less robust plans are likely created from rare problem cases where most rooms are already unlocked.

In conclusion, replanning consistently outperformed importance sampling regarding robustness. Especially in the case of the larger problem, we observed that replanning performed much better than importance sampling. Moreover, in these two problems, setting $\sigma$ to 0.25 yielded the best results. The modification of $\sigma$ also shares a similarity to the exploration-exploitation dilemma commonly seen in machine learning. If $\sigma$ was set too low, replanning did not frequently change its current plan. If $\sigma$ was set too high, it accepted too many plans which were not robust.

### 7.2 Limitations

Only two problems have been thoroughly investigated. The first problem is an extremely specific case and not representable of the possible challenges a traditional Keys and Doors problem may possess. There are only two directions the player could move, left and right. In addition, no keys and rooms were present in the first problem. However, this problem still shows the power that replanning has, since it is able to generate more robust plans than importance sampling by updating the plan for a problem where the previous plan failed. Other problems in the Keys and Doors domain may be so different from the two instances analysed, that replanning could show a much smaller or much greater performance difference.

## 8 Conclusions and Future Work

This research has aimed to answer whether the method in [1] by Gardos Reid can be generalized to work in different domains. The Keys and Doors problem has been used as the domain to explore the performance of the modified replanning approach. Several uncertainties were introduced in the problem, namely: start location, key location and unlocked rooms. The research has shown that the modified method, replanning, has a significantly better performance than importance sampling. The parameter $\sigma$ made a substantial difference in the performance of replanning and assigning $\sigma$ to 0.25 provided optimal results. Despite not being able to generate plans of 100% robustness, replanning has shown to be a promising technique to create robust plans, especially compared to importance sampling.

There are several areas in which research on replanning can be extended. First, a different probability distribution could be used instead of the currently employed normal distribution. The distribution could have a significant effect on performance, possibly even more substantial than tweaking $\sigma$. Second, simulated annealing can also be applied to replanning. Starting with a larger $\sigma$ and subsequently reducing it could solve the exploitation-exploration dilemma currently seen in the replanning approach. Third, it may be of interest to have plans of a maximum length. By adding a new random choice based on the length of the proposed plan, we can influence the length of proposals made by replanning. Lastly, the performance of replanning should be analysed in other domains, to validate and possibly improve its performance.

# References


[1] R. Gardos Reid, "Inferring robust plans with a rail network simulator", Master Thesis, Delft University of Technology, Delft, 2023, 62 pp. [Online]. Available: https://repository.tudelft.nl/record/uuid:209608e8-9fa7-4e03-aad8-e973ad22cde9.

[2] J.-W. v. d. Meent, B. Paige, H. Yang, and F. Wood, *An Introduction to Probabilistic Programming*. arXiv, Oct. 19, 2021. DOI: 10.48550/arXiv.1809.10756. arXiv: 1809.10756[stat]. [Online]. Available: http://arxiv.org/abs/1809.10756 (visited on 05/21/2025).

[3] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications", *Biometrika*, vol. 57, no. 1, pp. 97–109, Apr. 1970, ISSN: 0006-3444. DOI: 10.1093/biomet/57.1.97. [Online]. Available: https://doi.org/10.1093/biomet/57.1.97.

[4] T. Silver and R. Chitnis, *PDDLGym: Gym environments from PDDL problems*, Sep. 15, 2020. DOI: 10.48550/arXiv.2002.06432. arXiv: 2002.06432[cs]. [Online]. Available: http://arxiv.org/abs/2002.06432 (visited on 04/24/2025).

[5] E. Semenova, D. P. Williams, A. M. Afzal, and S. E. Lazic, "A bayesian neural network for toxicity prediction", *Computational Toxicology*, vol. 16, p. 100 133, Nov. 1, 2020, ISSN: 2468-1113. DOI: 10.1016/j.comtox.2020.100133. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2468111320300438 (visited on 06/02/2025).

[6] Z. Deng, D. Chu, C. Wu, *et al.*, "A probabilistic model for driving-style-recognition-enabled driver steering behaviors", *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 52, no. 3, pp. 1838–1851, 2022. DOI: 10.1109/TSMC.2020.3037229.

[7] A. Green. "What is PDDL?", Planning.wiki - The AI Planning & PDDL Wiki. (n.d.), [Online]. Available: https://planning.wiki/guide/whatis/pddl (visited on 06/19/2025).

[8] M. Ghallab, A. Howe, C. Knoblock, *et al.*, "PDDL | The Planning Domain Definition Language", Yale Center for Computational Vision and Control, Tech. Rep. CVC TR-98-003 / DCS TR-1165, 1998.

[9] M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. K. Mansinghka, "Gen: A general-purpose probabilistic programming system with programmable inference", in *PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Phoenix, Arizona: ACM, 2019, pp. 221–236.

[10] K. van der Knaap, *Robust plan inference in the keys and doors problem: Code*, Jun. 2025. DOI: 10.5281/zenodo.15711792. [Online]. Available: https://doi.org/10.5281/zenodo.15711792.

# A Custom Problem Replanning, $\sigma = 1.0$

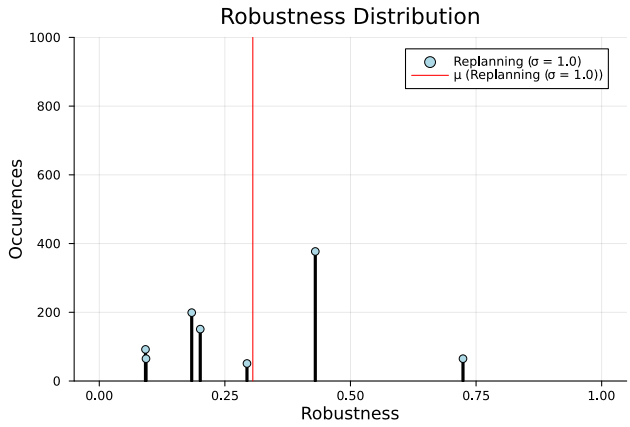



Figure 7: Distribution of robustness of plans when using replanning, $\sigma = 1.0$

# B Example Problem Replanning, $\sigma = 1.0$

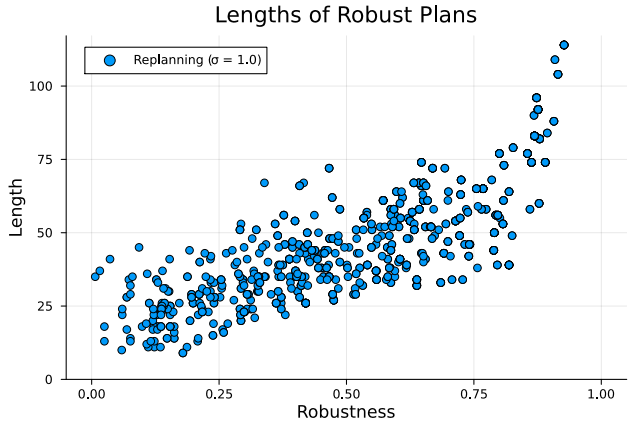



Figure 8: Length of plans shown against robustness of plans using Replanning, $\sigma = 1.0$

# C Usage of AI

A large language model has only been used to help writing code for converting between the `KeysAndDoors` struct, defined in Julia, and the `PDDLEnv` class, defined in Python. The language model used was `gemini-2.5-pro-preview-05-06`. The prompt supplied to Gemini is quite long due to included code. Code is left out for brevity and replaced by placeholders. The output itself is also long. The output is located in the function `to_python_pddl` in `problem_runner.jl`.

Could you create a method to convert from the KeysAndDoors struct to the PDDLEnv class. The code should be written in Julia. I have already made the code to convert from the PDDLEnv class to the KeysAndDoors struct. The code for this can be found here:

<Code converting PDDLEnv to KeysAndDoors>

The following code block specifies the PDDLEnv class:
<Code PDDLEnv class>

The following code block is the file containing the domain file of the keys and doors problem:
<Domain file>