CS 520: Assignment 3 - Probabilistic Search (and Destroy)

Haoyang Zhang, Han Wu, Shengjie Li, Zhichao Xu

November 22, 2018

1 Introduction, group members and division of workload

In this group project, we implemented a well-working solver that contains 5 searching strategies. This solver is also capable of dealing with a not-teleportable agent and a moving target. Our program also has a gorgeous GUI and can visualize the progress of solving by animation.

Name RUID	Workload
Haoyang Zhang 188008687	Implemented the searching solver. Wrote UpdateExplanation.html and RuleExplanation.html which are two documents about our algorithm, covering scores of important parts of our report (We put most of them into our report). Ran test for a moving target part.
Han Wu 189008460	Wrote the scripts to test the algorithm for question 4 in part one and moving target part. Ran tests together with Haoyang Zhang in moving target part. Generated most of the figures in the test. Wrote part of the report about question 4 in part one and the moving target part. Wrote answer for question 5.
Shengjie Li 188008047	Designed and implemented the GUI of our program. Implemented a function that can generate animation of the progress of searching and destroying. Finished the format design of the report using LaTeX.
Zhichao Xu 188008912	Helped revise and improve the codes through early-stage testing. Wrote scripts for experiments, did visualization and answered question 3.

2 A Stationary Target

2.1 Theory

2.1.1 Questions

1. Given observations up to time t (Observations$_t$), and a failure searching Cell$_j$ (Observations$_t + 1$ = Observations$_t \wedge$ Failure in Cell$_j$), how can Bayes' theorem be used to efficiently update the belief state, i.e., compute:

$$\mathbb{P}(\text{Target in Cell}_i|\text{Observations}_t \wedge \text{Failure in Cell}_j). \tag{1}$$

When $i \neq j$,

$$\begin{aligned}&\mathbb{P}(\text{Target in Cell}_i|\text{Observations}_t \wedge \text{Failure in Cell}_j)\\ &= \alpha\mathbb{P}(\text{Target in Cell}_i|\text{Observations}_t).\end{aligned} \tag{2}$$

Denote $\mathbb{P}$ (Target in Cell_i|Observations_t) by a_{it},

$(2) = \alpha \cdot a_{it}$.

When $i = j$,

$$\mathbb{P}(\text{Target in Cell}_j|\text{Observations}_t \wedge \text{Failure in Cell}_j)$$
$$= \alpha\mathbb{P}(\text{Target in Cell}_j|\text{Observations}_t) \cdot \mathbb{P}(\text{Target not found in Cell}_j|\text{Target in Cell}_j). \quad (3)$$

Denote $\mathbb{P}$ (Target not found in Cell_j|Target in Cell_j) by q_j,

$(3) = \alpha \cdot a_{jt} \cdot q_j$.

$$\alpha = \frac{1}{\sum_{i\neq j}(\alpha \cdot a_{it}) + \alpha \cdot a_{jt} \cdot q_j} = \frac{1}{1 - a_{jt}(1 - q_j)}.$$

$$a_{i0} = \frac{1}{2500} \text{ for } i = 0, \ldots, 2499$$

2. Given the observations up to time t, the belief state captures the **current probability the target is in a given cell**. What is the probability that the target will be **found** in Cell_i if it is searched:

$$\mathbb{P}(\text{Target found in Cell}_i|\text{Observations}_t)? \quad (4)$$

$$\mathbb{P}(\text{Target found in Cell}_i|\text{Observations}_t)$$
$$= \mathbb{P}(\text{Target in Cell}_i|\text{Observations}_t)(1 - \mathbb{P}(\text{Target not found in Cell}_i|\text{Target in Cell}_i))$$
$$= a_{it} \cdot (1 - q_i)$$

$$a_{i0} = \frac{1}{2500} \text{ for } i = 0, \ldots, 2499$$

2.1.2 Strategies of updating

1. Update search results
   For a given block, there are 2 different search results: 'True'(find the target) and 'False'(Target not found). If the result is 'True', this board is done. If the result is 'False', we have to update the 'prob' to represent this result.

   Using the notion of particle filter, assuming that each block have $N \cdot p_i$ samples. If we have not found a target in $block_k$, some samples should be cast off, i.e. $p'_k = p_k(1 - q_k)$. Then, resample all blocks by multiplying α, where $\alpha = \frac{1}{1-p_k q_k}$.

   In a nutshell, after searching $block_k$: $p'_i = \frac{\alpha N \cdot p_i}{N}$, where $i \neq k$; $p'_k = \frac{\alpha N \cdot p_k(1-q_k)}{N}$, $\alpha = \frac{1}{1-p_k q_k}$

   The pseudo code is as follows:

Algorithm 1 updateP(pos)

```
prob[pos] = prob[pos] * failP[cell[pos]]
normalize(prob)
return
```

Algorithm 2 normalize(prob)

```
normalize(prob):
sumP = sum(prob)
prob = prob / sumP
return
```

2. Update with additional observations
   Using the notion of particle filter, we can update reports easily. For a given cell, if it is not a possible previous block, all its samples should be ruled out. If it is a possible previous block, and if there are some possible neighbors according to the reports, its samples will migrate to them. But if there are no possible neighbors, all its samples will also be trapped in this block and die out. Then, resample all blocks.

   Yet, notice that reports are related to each other. For example, 2 consecutive reports H-F, C-H tell far more than "the target first move from Hilly to Flat or from Flat to Hilly, and then move form Cave to Hilly or from Hilly to Cave". We know that the target must move from Flat to Hilly and then to Cave.

   Therefore, we should consider each consecutive pair of reports and try to figure out the real moving direction. Once we know the real direction, we can pin down the direction of future reports, but also all previous reports.

   Notice that if we know the real direction of previous reports, we should re-filter all previous observations because observations have become more accurate.

   The pseudo code is as follows:

Algorithm 3 updateR(report)

```
reUpFlag, directions = solveReprot(report)
if reUpFlag then
  reUpdateReport(searchHistory, targetHistory)
else
  updateReport(directions)
end if
return
```

Algorithm 4 solveReport(report)

```
if reportSolved then
  direction = trackTarget(report, targetHistory[-1])
  reUpFlag = False
else
  common = findCommon(report, reportHistory[-1])
  if len(common) == 1 then
    backtrack(common, reportHistory)
    direction = tarckTarget(report, targetHistory[-1])
    reUpFlag = True
  else
    direction = (report, report.reverse())
    reUpFlag = False
  end if
end if
reportHistory.append(report)
return  reUpFlag, direction
```

Algorithm 5 reUpdateReprot(searchHistory, targetHistory)

```
for i in range(len(searchHistory)) do
  updateP(searchHistory[i])
  updateReport(((targetHistory[i], targetHistory[i+1]), ))
end for
return
```

Algorithm 6 updateReprot(directions)

```
tempProb = zeros_like(prob)
for prev, post in directions do
  for each block pos do
    if cell[pos] == prev then
      nPosList = where(border[pos, post])
      factor = len(nPosList)
      if factor then
        for each nPos in nPosList do
          tempPorb[nPos] = tempPorb[nPos] + prob[pos] / factor
        end for
      end if
    end if
  end for
end for
prob = normalize(tempProb)
return
```

Algorithm 7 trackTarget(report, prev)

```
post = report - prev
targetHistory.append(post)
return  post
```

Algorithm 8 backtrack(post, reportHistory)

```
targetHisory.insert(0, post)
for report in reportHistory.reverse() do
  prev = reprot - post
  targetHistory.insert(0, prev)
end for
return
```

2.1.3 Our understandings and interpretations of rules

Denote $P(target\ in\ block_i|Observation_t)$ by p_i.
Denote $P(find\ the\ target\ in\ block_i|target\ is\ in\ block_i)$ by q_i.

1. Rule 1
   A given rule in Assignment Description.
   - idea
     At any time, search the cell with the highest probability of containing the target.
   - Implementation
     $next = \arg\max_i p_i$
   - Pseudo Code

```
maxProb():
  value = prob
  return argmax(value)
```

2. Rule 2
   A given rule in Assignment Description.

   - idea
     At any time, search the cell with the highest probability of finding the target.
   - Implementation
     $next = \arg\max_i p_i q_i$
   - Pseudo Code

```
maxSucP():
  value = prob * sucP
  return argmax(value)
```

   - Discussion
     When there are many blocks, $p_i \ll 1$, while $C \approx \frac{0.2}{0.1} + \frac{0.3}{0.3} + \frac{0.3}{0.7} + \frac{0.2}{0.9} = 3.65$ Therefore, in order to minimize C_i', we can maximize α, which is maximize $p_i q_i$

3. Rule 3
   A rule just for fun.

   - idea
     At any time, search the block which provides the largest information entropy
   - Implementation
     $H_i = -(p_i q_i) log(p_i q_i)$ $next = \arg\max_i H_i$
   - Pseudo Code

```
maxInfo():
  find = prob * sucP
  value = find * log2(find)
  return argmin(find)

```

   - Discussion
     Notice that $H(x)$ is increasing function in $[0, 0.5]$. If there are a large number of blocks, Rule 3 is equivalent to Rule 2.

2.2 Practice

2.2.1 Questions

1. Consider comparing the following two decision rules:

   - Rule 1: At any time, search the cell with the highest probability of containing the target.
   - Rule 2: At any time, search the cell with the highest probability of finding the target.

   For either rule, in the case of ties between cells, consider breaking ties arbitrarily. How can these rules be interpreted / implemented in terms of the known probabilities and belief states?

   For a fixed map, consider repeatedly using each rule to locate the target (replacing the target at a new, uniformly chosen location each time it is discovered). On average, which performs better (i.e., requires less searches), Rule 1 or Rule 2? Why do you think that is? Does that hold across multiple maps?

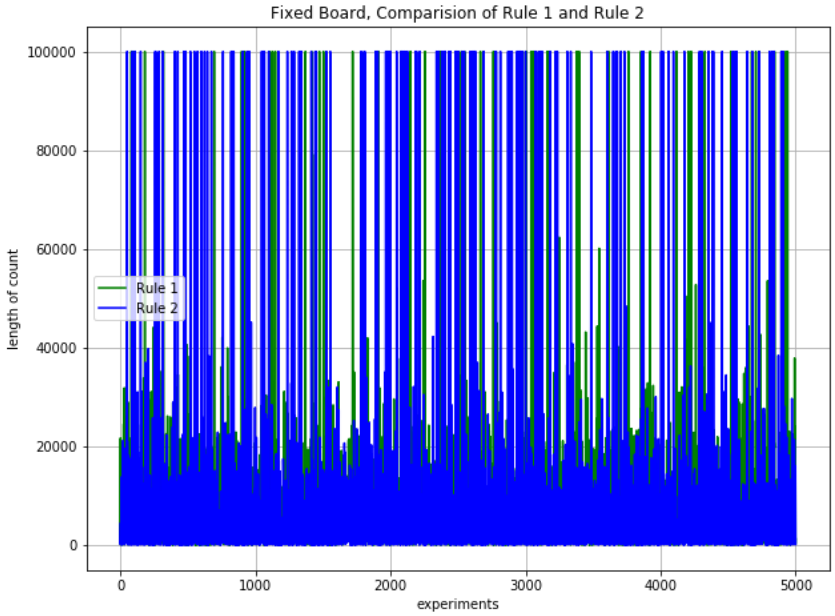

Figure 1: Fixed Board, Comparison of Rule 1 and Rule 2

For a fixed map, we repeatedly used Rule 1 and Rule 2 to locate the target. The result is shown in Fig 1.

The mean and variance of Rule 1 are 6004.0966 and 56320634.4832, and the mean and variance of Rule 2 are 4539.7006 and 37546839.3709 (after eliminating the unsolvable cases). The fail rate for Rule 1 and Rule 2 are 0.33% and 0.66%. These numbers are calculated through 5000 times of experiments. When the number of search actions exceeds 100000, we note the board as unsolvable. Through our comparison, we came into conclusion that Rule 2 is better than Rule 1. Also, it has larger fluctuations due to the presence of unsolvable cases.

According to our understanding, Rule 2 is more close to the problem itself, because we have $\mathbb{P}$(Target not found in Cell|Target is in Cell). The improvement of Rule 2 compared to Rule 1 is that Rule 2 gets rid of the conditional probability and make more accurate inferences.

For multiple maps, such pattern still holds. The result is showed in Fig 2.

The mean and variance of Rule 1 are 6221.3866 and 59873979.75314046, and the mean and variance of Rule 2 are 4682.4286, 40237182.382102035 (after eliminating the unsolvable cases). The fail rate for Rule 1 and Rule 2 are 0.33% and 0.66%. These numbers are also calculated through 5000 times of experiments.

In addition, we used a technique what we called “double check” to help searching. When we move to a flat cell, we would search it again if we have failed to find the target at the first search attempt. When we move to a hilly cell, if we have failed three search attempts on this cell, we would search it again. We call this “Double-check search” case. We use Rule 1, Rule 2, and Rule 3 to supervise our search. Rule 1 and Rule 2 is just the ones mentioned in the context. Even though Rule 3 does not have improvement, we also do some experiments because we find it interesting. In the experiment, 5000 maps are generated and we used 3 rules to search the target in the map. The result is shown in Fig 3.

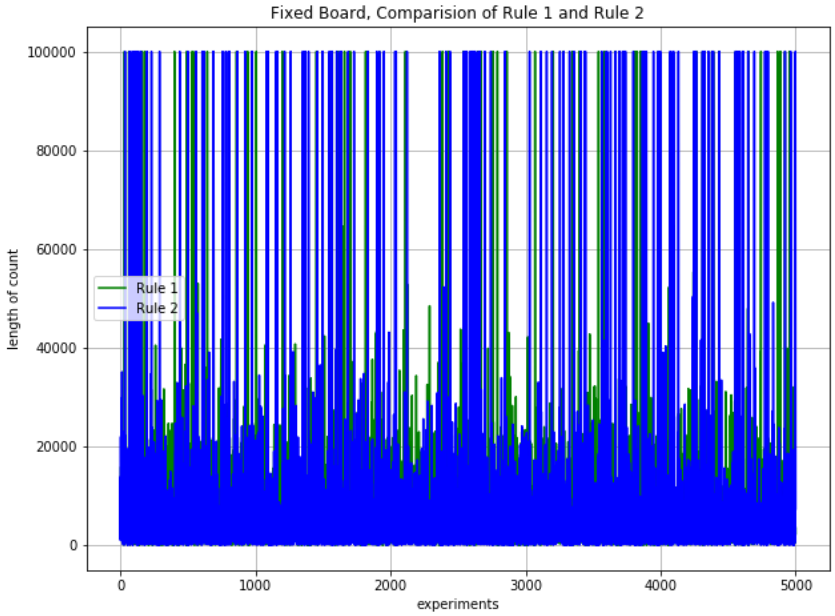

Figure 2: Fixed Board, Comparison of Rule 1 and Rule 2

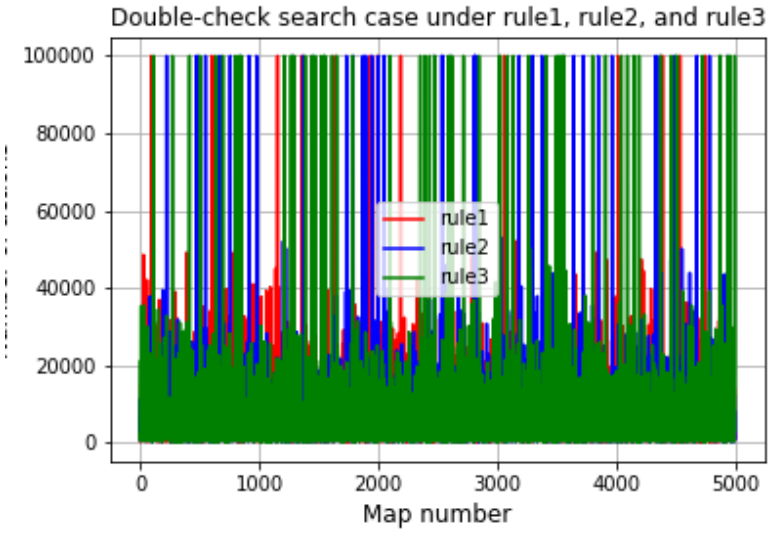

Figure 3: Number of actions in double-check search case

From the figure above, we can see that the performance of Rule 1 is the worst. Rule 2 and Rule 3 are similar. The means of Rule1, Rule 2, and Rule 3 are 6287.49, 5223.53, and 5145.19. The average number of actions of Rule 2 and Rule 3 are superior to Rule 1. So, Rule 2 and Rule 3 are better in this case. The variances of Rule1, Rule 2, and Rule 3 are 54138591.92, 42819190.19, and 39314588.23. From this perspective, we can also conclude that Rule 2 and Rule 3 are better. If the number of search actions is beyond 100,000, we call it unsolvable because it is extremely hard to find the target under this circumstance. The reason of it being unsolvable is that, if the target is on a 'Flat' cell or a 'Hilly' cell and we have failed several times searching on that cell, we would hardly be able to find the target in a reasonable time. To avoid the waste of time, we decided to terminate the program if the number of search action is greater than 100,000, which we call it unsolvable. The fail rate of Rule1, Rule 2, and Rule 3 are 0.0026, 0.008, 0.0116. Rule 3 may

sometimes not be able to find the target although it performs better.

2. Consider modifying the problem in the following way: at any time, you may only search the cell at your current location, or move to a neighboring cell (up/down, left/right). Search or motion each constitute a single 'action'. In this case, the 'best' cell to search by the previous rules may be out of reach, and require travel. One possibility is to simply move to the cell indicated by the previous rules and search it, but this may incur a large cost in terms of required travel. How can you use the belief state and your current location to determine whether to search or move (and where to move), and minimize the total number of actions required? Derive a decision rule based on the current belief state and current location, and compare its performance to the rule of simply always traveling to the next cell indicated by **Rule 1** or **Rule 2**. Discuss.

   In this question, we need to move to the cell that we want to search and motion also counted as one step. We call this "search and motion case". We use 3 rules to guide our search. Rule 1 means just traveling to the "best" cell which is indicated by the original Rule 1. Rule 2 is the same case. Rule 4 is invented by ourselves. It uses new criterion to supervise our search. We generate 5000 maps and use different rules to search the target in the maps. The result is shown in Fig 4.

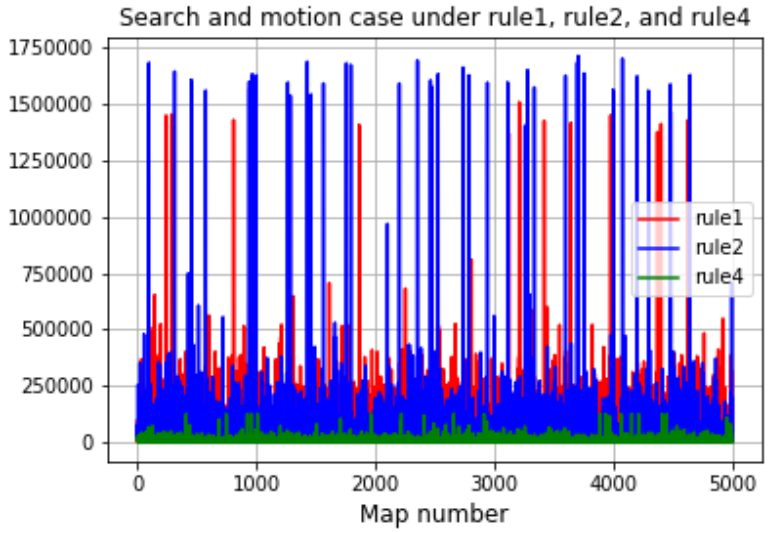

Figure 4: Number of actions in search and motion case

   From the figure, we can see that the performance of Rule 4 is better than Rule 1 and Rule 2. The means of Rule1, Rule 2, and Rule 4 are 58475.71, 45925.69, and 9745.66. It is easy to find that the average number of actions of Rule 4 is much smaller than Rule 1 and Rule 2. The variances of Rule1, Rule 2, and Rule 4 are 7920337425.95, 5069004957.26, and 119653468.99. From this view, we can also conclude that Rule 4 is better. When the number of search actions is beyond 100000, we call it unsolvable because it is extremely hard to find the target under this circumstance. The fail rate of Rule1, Rule 2, and Rule 4 are 0.0026, 0.0076, and 0.003. Rule 4 also performs well.

3. An old joke goes something like the following:

   A policeman sees a drunk man searching for something under a streetlight and asks what the drunk has lost. He says he lost his keys and they both look under the streetlight together. After a few minutes the policeman asks if he is sure he lost them here, and the drunk replies, no, and that he lost them in the park. The policeman asks why he is searching here, and the drunk replies, "the light is better here".

In light of the results of this project, discuss.

There are some links between the joke and our problem. There are also some differences between the drunk man in the joke and our search policy. Streetlight is a good thing. When you have streetlight, it is easier for you to find the keys than other area if they are actually under the streetlight. Flat area is just like the streetlight in the map. When the target is in one cell, you have the highest probability to find it if the cell is flat. So, you should search the flat area at first (and the result is: Rule 2 is better than Rule 1). However, searching the flat area is just one part of our policy. Another part is that we continuously update the probability. We continuously update our knowledge about the map. And this distinguishes our policy from the drunk man's. We update the probability of finding the target in one cell every time after we finish one search. And we search the cell with the highest probability of finding the target at the next time. When the flat cell has been searched, the probability of finding the target in the cell becomes very low. So, next time we will not choose to search it. However, the drunk man doesn't update his knowledge about the whole area. If he didn't find the keys under the streetlight, he should update his knowledge and go to other areas to search. It's unwise for him to keep searching under the streetlight.

Another problem of the drunk man is that he doesn't use his prior knowledge about the lost place. He remembers that he lost the keys in the park. So, he should search in the park at first although there may be no light. Actually, the drunk man's situation is better than us. In the map, every cell has the same probability of owning the target and we know nothing about the position of the target at first.

3 A Moving Target

In this section, the target is no longer stationary, and can move between neighboring cells. Each time you perform a search, if you fail to find the target the target will move to a neighboring cell (with uniform probability for each). However, all is not lost - whenever the target moves, surveillance reports to you that the target was seen at a **Type1** $\times$ **Type2** border where Type1 and Type2 are the cell types the target is moving between (though it is not reported which one was the exit point and which one the entry point.

Implement this functionality in your code. How can you update your search to make use of this extra information? How does your belief state change with these additional observations? Update your search accordingly, and again compare **Rule 1** and **Rule 2**.

The strategies of updating is in section 2.1.2.

Re-do question 4) above in this new environment with the moving target and extra information.

In this problem, the target is moving and we can search the cell that we are interested in directly. We don't need to move to the cell that we decide to search. We call this case "Moving case". We use Rule 1, Rule 2, and Rule 5 to supervise our search. Rule 1 and Rule 2 are the same as we mentioned before. Rule 5 is designed by ourselves. 5000 maps are generated and searched under 3 rules. The result is shown in Fig 5.

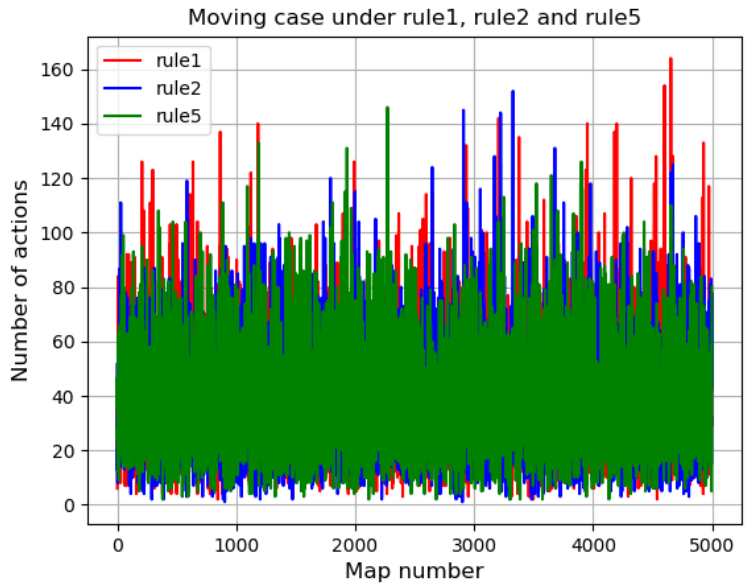

Figure 5: Number of actions in moving case

From the figure above, we find that the performance of 3 rules are similar, but Rule 1 has a minor disadvantage against Rule 2 and Rule 5. The means of Rule 1, Rule 2, and Rule 5 are 39.25, 37.82, and 38.04. The variances of Rule 1, Rule 2, and Rule 5 are 379.82, 347.52, and 351.06, from which we can see Rule 1 has a difference of 3% over the other two rules.

In this problem, the target is moving and motion also counted as one step. We call this case "moving and motion". We use three rules to guide our search — Rule 1, Rule 2, and Rule 5. Rule 1 and Rule 2 are just the original rules where motion is one step now. Rule 5 is designed for this specific problem. We generate 5000 maps and use these 3 rules to search the target in the maps. The result is shown in Fig 6.

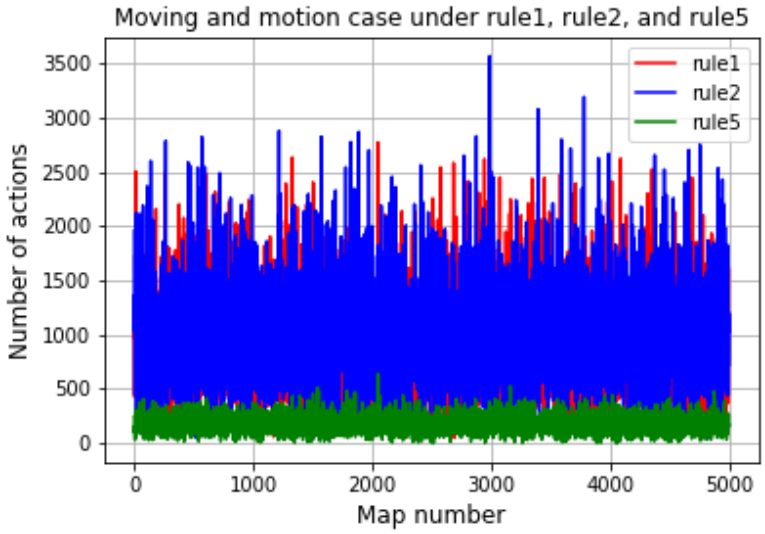

Figure 6: Number of actions in moving and motion case

From the figure above, it is easy to find that the performance of Rule 5 is better than Rule 1 and Rule

2. The means of Rule1, Rule 2, and Rule 5 are 938.88, 991.54, and 177.54. We can see that the average number of actions of Rule 5 is smaller than Rule 1 and Rule 2. The variances of Rule1, Rule 2, and Rule 5 are 154395.57, 176673.58, and 5534.82. From this view, we can also conclude that Rule 5 is better.