Assignment 0 (Warm-Up, Variant B): Clean the Wumpus Cave

AI-1 Systems Project (Winter Semester 2023/24)

Jan Frederik Schaefer

Friedrich-Alexander-Universität Erlangen-Nürnberg, Department Informatik

Topic:	Agents in AI, search
Due on:	March 15, 2024
Version from:	January 12, 2024
Author:	Jan Frederik Schaefer
Important notes:	To be solved individually
	Earlier deadline for first results (see your repository for submitting solution)

1 Task summary

A recurring theme in the AI lecture is the Wumpus world. The Wumpus is a mystical creature that lives in a cave that is organized as a grid of squares. We want to clean the Wumpus cave using a vacuum cleaner robot, which we can control with a sequence of instructions. You have to implement two tasks:

1. Check if a sequence of instructions cleans the entire Wumpus cave.
2. Come up with a sequence of instructions yourself – the shorter, the better.

The assignment repository [AR] contains files with problem representations that you have to solve. Your grade will largely be based on those solutions (see Section 7). The assignment repository also contains example solutions that you can use to test your implementation.

Didactic objectives

1. Develop an algorithm to solve a non-trivial problem,
2. implement a small software project from scratch,
3. get hands-on experience with a search problem,
4. improve the efficiency of an algorithm,
5. get to know the AISysProj setup and workflows.

Prerequisites and useful methods

1. The basics of computer science and programming,
2. Search (in a very general sense).

2 Maps

You have a **map** of the Wumpus cave, which consists of 18×12 squares. Figure 1 shows an example map. Every square has coordinates associated with it. As is common in computer science, the y-axis points down and the origin, $\langle 0, 0 \rangle$, is in the top-left square.

The properties of each square are represented by a single character:

1. Walls are marked with an `X`.
2. Empty squares are marked with a space.
3. The starting position of the vacuum cleaner (if it is known) is marked with an `S`.
4. If the initial orientation of the vacuum cleaner is also known, the starting position is instead marked with a
   (a) `^` to indicate that it faces up,
   (b) `>` to indicate that it faces right,
   (c) `v` to indicate that it faces down,
   (d) `<` to indicate that it faces left.
5. Portals (if they exist) are marked with `P`.

The maps are stored in the problem files (see Section 4) using a text representation: each row of the map corresponds to a line in the text representation and each square to a character. Figure 1 shows an example map with both the text representation and a more visual representation.

3 Plans and potentially missed squares

You can control the vacuum cleaner by making a **plan**, which is a sequence of instructions. The following instructions are available:

1. **`L`**: The vacuum cleaner turns 90° to the left.
2. **`R`**: The vacuum cleaner turns 90° to the right.
3. **`M`**: The vacuum cleaner moves 1 square in the direction it is currently facing.

We assume that the vacuum cleaner cleans every square it is on. If the instruction would move the vacuum cleaner onto a wall, it will instead remain on its current square. If the

```
XXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXX
XXXXXXXXXXX   XXXX
XXXXXXXXXX    XXXX
XXXXXXXXX  XXXXXXX
XXX  XXXX   XXXXXX
XX     XX  XXXXXXX
X   P       XXXXXX
XXP         XXXXXX
X          XXXXXXX
XXX S    XXXXXXXXX
XXXXXXXXXXXXXXXXXX
```

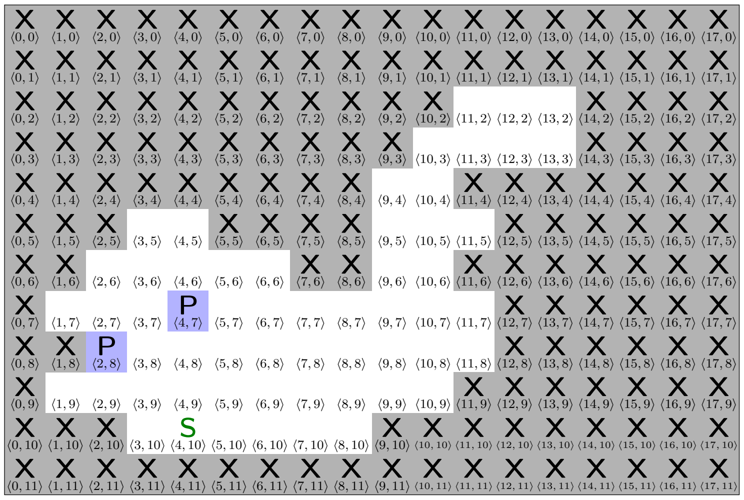


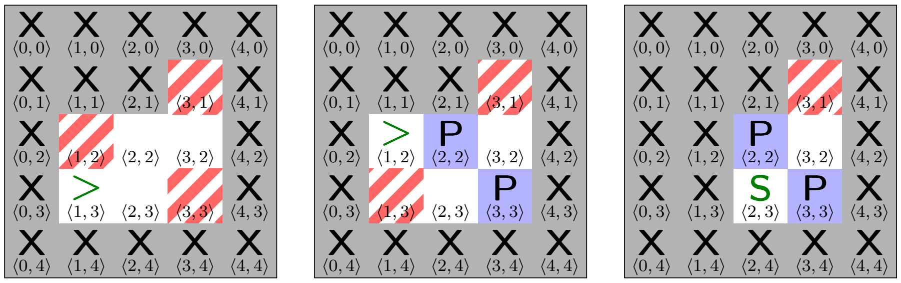
Figure 1: An example map of the Wumpus cave with the starting position marked at $\langle 4, 10\rangle$ and two portals marked with Ps. The text representation (left) is used in the problem files.

vacuum cleaner gets moved onto a portal square, it will immediately get teleported to the other portal square (there should always be exactly 2 portals). Note that portals do not have to get cleaned by the vacuum cleaner.

A good plan for the vacuum cleaner makes sure that everything gets cleaned. More concretely, we are interested in the **potentially missed squares**, which might still be dirty after executing the cleaning plan. To illustrate this, we will take a look at a few examples.

Example: Simple cave In this example, we will explore what happens if we execute the plan MLMMRM in the cave shown in Figure 2a. We start in $\langle 1, 3\rangle$ facing to the right. After executing M, we will be in $\langle 2, 3\rangle$. After executing L, we will face up, which means that after executing M, we will be in $\langle 2, 2\rangle$. If we execute M another time, we will remain in $\langle 2, 2\rangle$ because $\langle 2, 1\rangle$ is blocked. After executing R and M, we will be in $\langle 3, 2\rangle$. Therefore, we will not have cleaned $\langle 3, 1\rangle$, $\langle 1, 2\rangle$ and $\langle 3, 3\rangle$.

Example: Portals In this example, we will explore what happens if we execute the plan MMLMRRMM in the cave shown in Figure 2b. We start at $\langle 1, 2\rangle$ and face east. Walking one step forward, we land on the portal at $\langle 2, 2\rangle$, which immediately teleports us to $\langle 3, 3\rangle$. Now there is a wall in the east, which means that we will stay at $\langle 3, 3\rangle$ for the next M action. Then we go north (actions LM, landing on $\langle 3, 2\rangle$) and south again (RRM), which means we will teleport from $\langle 3, 3\rangle$ to $\langle 2, 2\rangle$. We will still face south, so the last M action leads us to



(a) plan MLMMRM (b) plan MMLMRRMM (c) plan MMLMMLMM

Figure 2: Example caves. The potentially missed squares are marked with diagonal lines (▨).

$\langle 2,3\rangle$. Therefore, we will clean $\langle 1,2\rangle$, $\langle 2,3\rangle$ and $\langle 3,2\rangle$. We will miss the squares $\langle 1,3\rangle$ and $\langle 3,1\rangle$ (the portals do not have to be cleaned).

Example: Portals and unknown initial orientation In this example, we will explore what happens if we execute the plan MMLMMLMM in the cave shown in Figure 2c. If we initially face east or south, we will clean the entire cave. However, if we initially face north or west, we will miss $\langle 3,1\rangle$. Therefore, we will potentially miss square $\langle 3,1\rangle$.

4 Problem and solution files

The assignment repository [AR] contains many **problem files**. Your implementation is supposed to generate a **solution file** for each problem file. This section describes the format of problem and solution files.

4.1 Checking plans

The easier problem files require you to check a cleaning plan. They begin with the line CHECK PLAN, followed by a plan as described in Section 3, followed by the text representation of a map as described in Section 2.

If there are no potentially missed squares, the solution file should contain the text GOOD PLAN. Otherwise, the solution file should contain the text BAD PLAN, followed by a list of the potentially missed squares (the order does not matter). For example, if the squares

$\langle 2, 3\rangle$ and $\langle 1, 5\rangle$ are potentially missed, the solution file should be

```
BAD PLAN
2, 3
1, 5
```

4.2 Finding plans

The more difficult problem files require you to find a cleaning plan. They begin with the line FIND PLAN, followed by the text representation of a map. The solution file should then contain the plan as described in Section 3.

If the format is not clear, you can take a look at the assignment repository [AR], which contains example problems and solutions.

Important: The number of points for plan finding problems depends on the plan lengths (see Section 7 for details).

5 What to submit

Your solution should be pushed to your git repository for this assignment. For this warm-up assignment, we have an early deadline, which should be in the README of your repository. At this deadline, the repository should contain:

1. all the code you have so far,
2. solutions to the problem_a_*.txt files.

Otherwise, we might assume that you are not actually interested in the project and give your spot to someone else.

Your grade will be based on your final submission (deadline: March 15, 2024). Concretely, your repository should contain:

1. all your code for solving this assignment,
2. a README.md file explaining how to run your code (including e.g. dependencies that have to be installed),
3. a brief evaluation either as a PDF file ($\approx \frac{1}{2}$ page, but it may be longer) or as part of your README.md, describing your approach for solving the problem and what worked well/did not work well,
4. solution files (as described in Section 4) for the problem files. The solution file for problem_X_YZ.txt should be called solution_X_YZ.txt.

6 A few tips

1. If you do not really understand what you are supposed to do, please consider asking about it in the office hours. This is fairly common and it would be a pity if you drop out just because you are a bit lost in the beginning. The project is probably quite different from the courses you are used to.
2. If you do not really know where to start, you might want to follow the "guide for getting started" (Appendix A).
3. Start with the simpler problems (in particular, the plan checking).
4. Finding plans is not easy. Here are a few tips to help you get started:
   (a) Do not try to find optimal (shortest) plans. It is too difficult for the larger caves.
   (b) A very simple starting point is to make a random plan: if you have a long sequence of randomly selected instructions, it is probably a valid plan. Then you can try to think of ways to make the generated plan shorter. You can also try to modify the random generation in a way that it tends to select instructions that will lead to a shorter plan. How could you identify "good instructions" to continue your plan?
   (c) Another starting point is to build up a plan step by step and always try to do something that gets you a little closer to the goal (e.g. clean one more square).
   (d) If you solve the example problems, you can use a script from the assignment repository [AR] to check if they are correct.
5. Use your tools for checking plans to make sure that the plans you find are correct.
6. Efficiency matters for these problems, but you do not have to use an extremely fast programming language (e.g. Python is completely sufficient). What really matters is the asymptotic time complexity.
7. Implement a loop for solving all problem files (instead of manually changing the file name in the code).

7 Points

The total number of points for this assignment is 100. You can get up to 20 points for the quality of the submission (README, evaluation, ...).

The remaining 80 points are awarded for the solutions to the problem files. Figure 3 shows how many points can be achieved for each part. For the FIND PLAN problems, the number of points depends on the total plan length T of your solutions, i.e. T is the sum of the lengths of the plans you found for that part.

Note that partial points (for solving only part of a problem range correctly) are only awarded in exceptional cases.

If the grading scheme doesn't seem to work well, we might adjust it later on (likely in your favor).

Important: You get points for *correct* solutions. You generally do not e.g. get partial points for code that "looks roughly correct but produces wrong results". The assignment repository contains a script that you can use to check your solutions for the example problems.

References

[AR] *Repository for Assignment 0 (Warm-Up, Variant B): Clean the Wumpus Cave.* URL: `https://gitlab.rrze.fau.de/wrv/AISysProj/ws2324/a1.0.b-clean-wumpus-cave/assignment`.

Problems	**Mode**	**Challenges**	**points**
problem_a_∗.txt	check	—	10
problem_b_∗.txt	check	portals	10
problem_c_∗.txt	check	unknown orientation portals	10
problem_d_∗.txt	find	—	$\begin{cases} 15 & \text{if } T \leq 20000 \\ 10 & \text{if } T \leq 35000 \\ 5 & \text{if } T \leq 60000 \\ 0 & \text{if } T > 60000 \end{cases}$
problem_e_∗.txt	find	portals	$\begin{cases} 15 & \text{if } T \leq 30000 \\ 10 & \text{if } T \leq 50000 \\ 5 & \text{if } T \leq 60000 \\ 0 & \text{if } T > 60000 \end{cases}$
problem_f_∗.txt	find	unknown orientation portals	$\begin{cases} 20 & \text{if } T \leq 40000 \\ 15 & \text{if } T \leq 50000 \\ 10 & \text{if } T \leq 60000 \\ 5 & \text{if } T \leq 80000 \\ 0 & \text{if } T > 80000 \end{cases}$

Figure 3: Points per part.

A Guide for getting started (optional)

The "guide for getting started" is a new attempt to support students with less programming experience. Concretely, we want to guide you with a series of comments and questions towards a (partial) solution and give you an idea how larger programming problems can be tackled. If you think that you do not need it, you can simply ignore it.

Please let us know if you find this guide useful (then we might provide it in future assignments as well)! We also appreciate if you suggest improvements.

A.1 How to use this guide?

This guide poses a lot of questions. You should try to think about them for yourself and implement some code for it – our hope is that that process will help you get started.

For example, if there is a question "what squares are adjacent to $\langle 3, 5\rangle$?" you could create the following (Python) code:

```python
def get_neighbours(x, y):
    return [(x+1, y), (x-1, y), (x, y-1), (x, y+1)]
print(get_neighbours(3, 5))
```

Of course there are many other ways (e.g. creating a Square class with a get_neighbours method). However, doing

```python
print([(4, 5), (2, 5), (3, 4), (3, 6)]
```

or

```python
print([(3+1, 5), (3-1, 5), (3, 5-1), (3, 5+1)])
```

would not be enough because it does not solve the problem of finding adjacent squares in general.

Usually, it is a good idea to test your implementations with more examples to make sure that you cover all the cases. This guide tends to intentionally skip over special cases so that you will discover them at some point yourself, which we consider a valuable teaching moment.

A.2 Overall approach

There are different ways to approach an assignment like this one. Here, we will first load the data from a problem file and afterwards try to work on the actual problem solving.

Another option would be to hard-code a map and a plan in the source code and implement the problem solving part before trying to actually load problem files.

A.3 Loading a problem file

Goal: load the data from a problem file into a representation that is easy to work with. The format of a problem file is described in Section 4. "Inspirational" questions and comments for you to think about and write some code for (see also Appendix A.1):

1. Does problem_a_04.txt require you to check a plan or to find one? (check the first line, and remember that you should create code for answering the question)
2. What is the plan that has to be checked in problem_a_03.txt?
3. What is the text representation of the map in problem_a_03.txt? (This could e.g. be a string or a list of strings.)
4. Try to think of a good way to represent the map. There are many ways to do this. You could even keep it as a string.
5. Load the map from problem_a_05.txt into that representation.
6. Is square $\langle 4, 2\rangle$ from problem_a_05.txt blocked? (Test this thoroughly by checking more squares – if you make a mistake, it might get very confusing later on.)
7. Design your code in a way that it is very easy to check if a square $\langle x, y\rangle$ is blocked (we might have to do that a lot).
8. Does problem_a_08.txt have a starting position?
9. What are the coordinates of the starting position in problem_a_08.txt?
10. What is the initial orientation in problem_a_08.txt?

At this point you should be able to load a problem file with your code and access the map data conveniently. With that in place, we can try to solve the first problem files.

A.4 Solving part a

Goal: solve the first problems. "Inspirational" questions and comments:

1. What square do I land on if I walk one square (action M) from square $\langle 2, 8\rangle$ when if I initially face up? (Try this with other directions and make sure there are no bugs.)
2. Answer the same question for different squares, but this time use a map (e.g. from problem_a_00.txt) and only change the position if the new square is free.
3. Make sure that everything from Appendix A.3 is ready:
   (a) What is the plan in problem_a_00.txt?

   (b) What is the starting position in problem_a_00.txt?
   (c) What is the initial orientation in problem_a_00.txt?
4. What is the first instruction of the plan in problem_a_00.txt?
5. What happens after executing the first action of the plan in problem_a_00.txt?
6. What squares do you cover when following all the instructions of problem_a_00.txt?
7. What are the free squares in problem_a_00.txt?
8. Are all the free squares covered by the instructions in problem_a_00.txt?
9. What squares are not covered by the instructions in problem_a_00.txt?
10. Create a solution file solution_a_00.txt for problem_a_00.txt (see Section 4).

A.5 Automation

You should now be able to create solution files for the first problem files. If you haven't done so yet, it is a good idea to automate everything as much as possible. Basically, you want to have code that will automatically create all the solution files. You should not have to change file paths in your script to solve different problems or manually copy-paste the output into a solution file.

The assignment repository contains example problems files with their solutions. It makes sense to have a tool that automatically compares all your solutions to the example solutions to help you find and debug potential bugs.

A.6 Continuing

We will not provide detailed instructions for the next problems, but we hope that the previous sections helped you get started. Section 6 might contain some helpful tips. If you feel like you need some more guidance, you are invited to come to the office hours or ask in the matrix room for this assignment.