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

AI-1 Systems Project (Winter Semester 2022/2023)

Jan Frederik Schaefer

Friedrich-Alexander-Universität Erlangen-Nürnberg, Department Informatik

Topic:	Agents in AI, search
Due on:	November 15, 2022
Version from:	October 24, 2022
Author:	Jan Frederik Schaefer

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 assignment repository [A0] contains files with problem representations that you have to solve. It also contains example solutions to test your implementation.

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. Nothing specific, just the basics of computer science and programming.

Copyright © 2021–2022 Jan Frederik Schaefer, Michael Kohlhase.

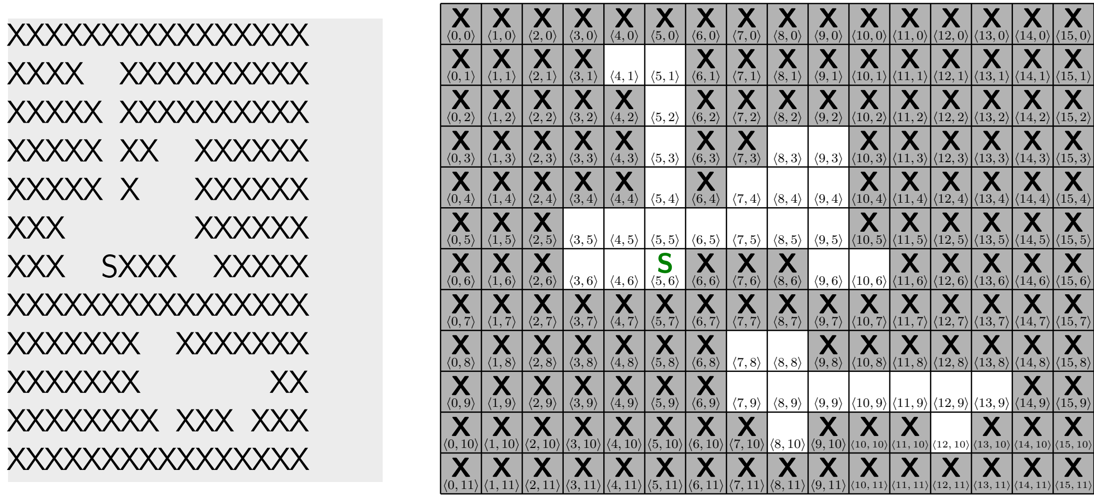


Figure 1: An example map of the Wumpus cave with the starting position marked at $\langle 5, 6 \rangle$. The text representation (left) is used in the problem files. In this example, the cave consists of two disconnected parts (rooms).

2 Maps

You have a **map** of the Wumpus cave, which consists of 16×12 squares. Each square is either **blocked** (by walls etc.) or **free**. If the starting position of the vacuum cleaner is known, it is marked with an S. The starting position is always a free square. For simplicity, the squares on the edge are always blocked.

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 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. Blocked squares are represented by an X and free squares by a space, except for the starting position, which is marked with an S (if known). Figure 1 shows an example map with both the text representation and a more visual representation.

The more advanced problem files can have a cave with disconnected parts (like the one shown above). We will call each connected component a **room** (for example, the cave in Figure 1 has two rooms).

3 Plans and potentially missed squares

A **plan** is a sequence of instructions for the vacuum cleaner. There are four instructions

1. **N**: The vacuum cleaner moves one square *north* (up).
2. **E**: The vacuum cleaner moves one square *east* (right).
3. **S**: The vacuum cleaner moves one square *south* (down).
4. **W**: The vacuum cleaner moves one square *west* (left).

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

A good plan for the vacuum cleaner makes sure that all the squares in the room get cleaned. More concretely, we are interested in the **potentially missed squares**, which we define to be any square P that fulfills two conditions:

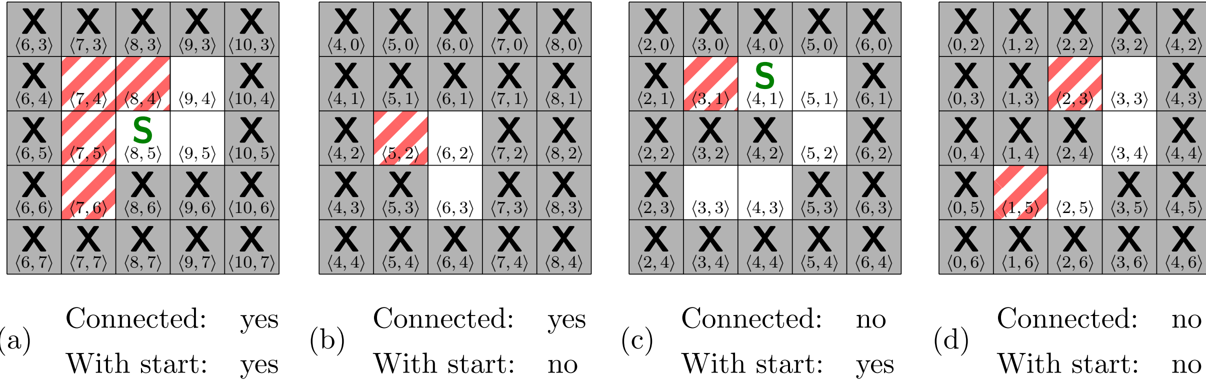
1. P belongs to a room R in which the vacuum cleaner might start,
2. if the vacuum cleaner starts in R, it is possible that it will not clean P with the given plan.

This definition is important but somewhat complicated, so let us take a look at a few examples.

Example: Fully-connected cave with starting position Let us evaluate the plan EENS for the example cave shown in Figure 2a. The vacuum cleaner starts on $\langle 8, 5\rangle$. The first instruction, E, moves the vacuum cleaner to $\langle 9, 5\rangle$. The second instruction, also E, would move the vacuum cleaner to a blocked square, so it is ignored. Afterwards, the vacuum cleaner moves to $\langle 9, 4\rangle$ and then back to $\langle 9, 5\rangle$. Therefore, the potentially missed squares are $\langle 7, 4\rangle$, $\langle 8, 4\rangle$, $\langle 7, 5\rangle$, and $\langle 7, 6\rangle$.

Example: Fully-connected cave without starting position Let as assume that we have the cave shown in Figure 2b without a starting position. With the plan EENS, $\langle 5, 2\rangle$ is a potentially missed square because it would not get cleaned if the vacuum cleaner starts on square $\langle 6, 2\rangle$ or $\langle 6, 3\rangle$.

Example: Disconnected cave with starting position The disconnected cave shown in Figure 2c has two rooms. Since we know that the vacuum cleaner will start in the upper room (because the starting position is marked), potentially missed squares can only be in that room. With the plan EENS, the only potentially missed square would therefore be $\langle 3, 1\rangle$.



(a) Connected: yes With start: yes (b) Connected: yes With start: no (c) Connected: no With start: yes (d) Connected: no With start: no

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

Example: Disconnected cave without starting position Figure 2d shows a cave with two rooms and without any marked starting position. If we start in the room on the upper right, we could miss square $\langle 2, 3\rangle$ (for example if we start in $\langle 3, 3\rangle$). And if we start in the other room, we could miss square $\langle 1, 5\rangle$.

4 Problem and solution files

The assignment repository [A0] 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. For example, the problem file for the example shown in Figure 2d would start with the lines

```
CHECK PLAN
EENS
XXXXXXXXXXXXXXXX
...
```

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 the example in Figure 2d, the solution file should contain the following text:

```
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.

Taking the example from Figure 2d again, the problem file would start with the lines

```
FIND PLAN
XXXXXXXXXXXXXXXX
...
```

and the solution file should contain the plan in a single line:

```
ESNW
```

> **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 gitlab repository for this assignment. Concretely, the 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 summary of how you solved the problem either as a PDF file ($\approx \frac{1}{2}$ page) or as part of your README.md,
4. solution files (as described in Section 4) for the chosen problem files. Note that you can choose between problem080.txt – problem119.txt and problem200.txt – problem219.txt (the latter might be easier but only gives partial points, see Section 7 for details). The solution file for problemxyz.txt should be called solutionxyz.txt.

6 A few tips

1. If you do not really know where to start, you might want to follow the "guide for getting started" (Appendix A).
2. Start with the simpler problems (in particular, the plan checking).
3. 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 examples.
   (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).
4. Use your tools for checking plans to make sure that the plans you find are correct.
5. 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 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, explanation, ...). The remaining 80 points are awarded for the solutions to the problem files. The subsection below describes how many points are awarded for what problem range. The details are rather complex because we want to give fair grades to students with very different skills.

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

7.1 Breakdown of partial points for the solution

You can get up to 40 points for checking plans. Concretely, you get 10 points for each of the following problem ranges (if solved correctly):

- problem000.txt – problem019.txt (connected cave with starting position)
- problem020.txt – problem039.txt (connected cave without starting position)
- problem040.txt – problem059.txt (disconnected cave with starting position)
- problem060.txt – problem079.txt (disconnected cave without starting position)

You can also get up to 40 points for finding plans. The points for a problem range depend on the sum of plan lengths for that range. For each of the following problem ranges you get 20 points if the sum of plan lengths is $< 3\,000$, 15 points if it is $< 10\,000$ or 10 points if it is $< 30\,000$ and 0 points if it is $\geq 30\,000$:

- problem080.txt – problem099.txt (connected cave without starting position)
- problem100.txt – problem119.txt (disconnected cave without starting position)

If you cannot solve the problem ranges for finding plans, you can instead solve

- problem200.txt – problem219.txt (connected cave with starting position)

with the same grading scheme as above. Depending on your approach, it might not be any easier though. In other words, you can either solve problem080.txt – problem119.txt (and get up to 40 points for finding plans) or solve problem200.txt – problem219.txt (and get only up to 20 points for finding plans).

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

References

[A0] *Assignment 0*. URL: `https://gitlab.rrze.fau.de/wrv/AISysProj/ws2223/a0-clean-the-wumpus-cave/assignment` (visited on 10/24/2022).

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 can 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:

```
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

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

or

```
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 problem020.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 problem003.txt?
3. What is the text representation of the map in problem003.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 problem005.txt into that representation.
6. Is square $\langle 4, 2\rangle$ from problem005.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 problem008.txt have a starting position?
9. What are the coordinates of the starting position in problem008.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 problem000.txt – problem019.txt

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

1. What square do I land on of if I walk N (north) from square $\langle 2, 8\rangle$? (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 problem000.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 problem000.txt?
   (b) What is the starting position in problem000.txt?

4. What is the first instruction of the plan in problem000.txt?
5. What square do you land on after the first instruction of the plan in problem000.txt?
6. What squares do you cover when following all the instructions of problem000.txt?
7. What are the free squares in problem000.txt?
8. Are all the free squares covered by the instructions in problem000.txt?
9. What squares are not covered by the instructions in problem000.txt?
10. Create a solution file solution000.txt for problem000.txt (see Section 4).

A.5 Checking the solutions

You should now be able to create solution files for the first 20 problem files. While you can check a few solutions manually, it will get very tedious. The assignment repository contains example problems files with their solutions. It makes sense to have a tool that automatically compares your solutions to the example solutions. "Inspirational" questions:

1. Does example-solution000.txt indicate that the plan in example-problem000.txt is good or bad?
2. What squares are marked as potentially missed in example-solution000.txt?
3. Does your solution to example-problem000.txt match the official example-solution000.txt?

A.6 Continuing

We will not provide detailed instructions for the next problems, but we hope that the previous sections helped you get started.

Nevertheless, here are a few remarks that you might find helpful:

1. The key challenge for problem040.txt – problem059.txt is to find all the squares that belong to the room with the starting position. Once you have solved that, it should be easy.
2. Similarly, for problem060.txt – problem079.txt you need to find all rooms (and what squares belong to them).
3. As mentioned in Section 6, an easy way to find a plan is to simply generate a random one and use your existing code to check if it is a good one. Then you can try to improve on that.