The University of Melbourne
School of Computing and Information Systems

Declarative Programming
COMP90048

Semester 1

Project Specification

Project due Thursday, 24th May 2018 at 5pm
Worth 15%

The objective of this project is to practice and assess your understanding of logic programming and Prolog. You will write code to solve a small planning problem.

Wumpus

Wumpus is a planning problem. You need to find and kill a Wumpus hiding in an unknown maze. The player sends in a series of disposable robots each with a fixed list of instructions to follow. Each robot follows its instructions until it is destroyed or it finishes the instructions, or it kills the Wumpus. A
sensed following the in
Wumpus.

The robots are very simple, they follow a fixed list of orders made up of

`north` move north: $(x, y) \rightarrow (x, y - 1)$

`east` move east: $(x, y) \rightarrow (x + 1, y)$

`south` move south: $(x, y) \rightarrow (x, y + 1)$

`west` move west: $(x, y) \rightarrow (x - 1, y)$

`shoot` shoot in the direction of the last move (if no moves this direction is north).

The map is a rectangular grid made up of 4 kinds of square

`empty` an empty space `.`

`pit` a deadly pit `P`

`wall` a solid block of wall `#`

`wumpus` the Wumpus (there is only one) `W`.

As the robot moves it gets feedback

- if it moves onto a `pit` it gets feedback `pit`, and falls to its death
- if it (attempts to) move onto a `wall` it gets feedback `wall` and stays in the same position
- if the robot tries to move off the map it gets feedback `wall` and doesnt move
- if it moves onto a `wumpus` it gets feedback `wumpus`, and is promptly eaten.

- if it moves onto an `empty` space it gets different possible feedback
  - if it is directly adjacent (not diagonally) to the `wumpus` it gets feedback `stench`, the strong odor of the Wumpus
  - if it is not adjacent but within 3 spaces (Manhattan distance) of the `wumpus` it gets feedback `smell`, weaker odor of the Wumpus
  - if it is further than 3 spaces from the Wumpus and directly adjacent (not diagonally) to one or more `pits` it gets feedback `damp`, the smell of the pit
  - if it is further than 3 spaces from the Wumpus and not directly adjacent to a pit it gets feedback `empty`

The robot shoots in the direction it last moved. The arrow flies until it hits a wall (including off the map) or the Wumpus. In the first case the feedback is `miss`, otherwise `hit`.

Consider the small map below on the left:

```
.......          xxxx...
.#P..W.          x#Px.x.
.#.....          x#.xxx.
```

A robot starting at position (1,3), the bottom left corner following instructions [`east`, `south`, `west`, `north`, `east`, `north`, `east`, `east`, `shoot`, `east`, `shoot`, `south`, `south`, `east`, `east`, `east`, `north`, `north`, `west`] it gets feedback [`wall`, `wall`, `wall`, `empty`, `wall`, `empty`, `empty`, `damp`, `miss`, `smell`, `mis` he feedback stops when it dies. The path it a is useless: it travels the sa shot from, so given the first shot didnt hit the Wumpus, this shot cannot either.

The aim is to kill the Wumpus in the least number of attempts. A to minimize the total energy required, but this is less important tha The energy required for an attempt is given by the sum of the mov of `shoot` actions. There is also an energy limit on any sequence of commands of 100.

The Program

You will write Prolog code to hunt the Wumpus the game. This will require you to write a functions to initialize some internal state, make a guess given the state, and update the state given the feedback. The last two functions will be called repeatedly until you kill the Wumpus. You may use any representation you like for internal state

You must define following functions:

initialState(+NR, +NC, +XS, +YS, -State0)
: takes as input the number of rows NR and number of columns NC in the game, and the starting position (XS, YS) and outputs $State0$ an initial state representation for the game.

guess(+State0, -State, -Guess)
: Given the current state $State0$ returns a new state (it could be the same as the current state) and a *Guess* which is a list of `north`, `east`, `south`, `west`, `shoot` which are instructions for the robot. The total energy of the *Guess* must be at most 100 energy: each move costs 1 energy, and each `shoot` requires 5 energy.

updateState(+State0, +Guess, +Feedback, -State)
takes as input the state *State*0, the previous guess *Guess* and the feedback from the guess *Feedback* and returns a new updated state *State*.

You must call your (only) source file `wumpus.pl` and it must contain the module declaration:

```
:- module(wumpus,[initialGuess/5, guess/3, updateState/4]).
```

All your code should be in a single file.

I will post a test driver program `wumpusrun`, which will operate similarly to how I actually test your code. You can test your code using the command:

```
wumpusrun map xs ys
```

which takes the name of a *map* file, and the (xs, ys) start position for the robots. This will automatically load your file `wumpus.pl`, assuming its in the same directory. You can compile `wumpusrun` on your own machine using

```
swipl --goal=go -o wumpusrun -c wr.pl
```


You will be given a set of sample maps, some of which will be used in the evaluation of the submission. The maps will be graded from easy to vhard (very hard), to give you some ideas about what you c

file is simple a rectangu

h exactly one `W`

Assessment

Your project will be assessed on the following criteria:

70% Quality and correctness of your implementation;

30% Quality of your code and documentation

The correctness of your implementation will be assessed based on how many maps it succeeds to kill the Wumpus, and on how many attempts it requires to kill on each map.

Note that timeouts will be imposed on all tests. You will have at least 10 seconds to find and kill each Wumpus, regardless of how many attempts are required. Executions taking longer than that may be unceremoniously terminated, leading to that test being assessed as failing. 10 seconds per test is a very reasonable limit.

See the Project Coding Guidelines on the LMS for detailed suggestions for coding style. These guidelines will form the basis of the quality assessment of your code and documentation.

Submission

You must submit your project from the student unix server `dimefox.eng.unimelb.edu.au` or `nutmeg.eng.unimelb.edu.au`. Make sure the version of your program source files you wish to submit is on this server, then `cd` to the directory holding your source code and issue the command:

```
submit COMP90048 project4 wumpus.pl
```

If your code spans multiple source files, add the extra ones to the end of that command line.

Important: you must wait a minute or two (or more if the servers are busy) after submitting, and then issue the command

```
verify COMP90048 project4 | less
```

This will show you the test results from your submission, as well as the file(s) you submitted. If the test results show any problems, correct them and submit again. You may submit as often as you like; only your final submission will be assessed.

If you wish to (re-)submit after the project deadline, you may do so by adding "`.late`" to the end of the project name (*i.e.,* `project4.late`) in the `submit` and `verify` commands. But note that a penalty, described below, will apply to late submissions, so you should weigh the points you will lose for a late submission against the points you expect to gain by revising your program and submitting again.

It is your responsibility to verify your submission.

Windows users should see the LMS Resources list for instructions for downloading the (free) Putty and Winscp programs to allow you to use and copy files to the department servers from windows computers. Mac OS X and Linux users can use the `ssh`, `scp`, and `sftp` programs that come with your operating system.

Late Penalties

Late submissions will l
of that submission per hour late, including evening and weekend hours. This means that a perfect project that is 30 seconds late will lose 20% and one that it is tw
68% of the marks for the project. If you have a medical or similar c
late, you should contact the lecturer as early as possible to ask for an e
before the due date).

Hints

1. The key to this project is that every guess *must* gain new information. Your robot instructions must explore the maze, find the Wumpus, and then get a robot to move to shoot the Wumpus. Every attempt must be guaranteed to gain new information about the map, otherwise you can get stuck in a loop.

2. A critical decision for the project is your representation of state. You need to track what parts of the map you have visited. You also may want to track which shots (which position and direction) you have taken. You may want to add more state as you get a better solution.

3. The first approach should be to ignore the `damp`, `smell` and `stench` feedback and simply concentrate on recording accurate information about the map.

4. You need to be careful that when searching for a plan that the search doesnt go into an infinite loop. You can prevent this by making sure that plan uses at most 100 energy.

5. The key part of your code is to use the ability of Prolog to search for a possible plan. The plan must find new information. You may break down the project into two parts: e.g. plans to find the Wumpus, and then plans to shoot the Wumpus. Or you might try to do everything at once, shooting wildly in all directions as you traverse the maze.

   You can use your `state` type to store a representation of the map, a representation of shots you have previously taken, a representation of where the Wumpus could possibly be or not be.

6. Once you have a basic version that works you can consider things like: making very long plans that generate lots of information about the map; making plans that narrow down the position of the Wumpus quickly; making plans that only shoot where a previous shot could not have reached the Wumpus.

7. Note that these are just hints; you are welcome to use any approach you like to solve this, as long as it is correct and runs within the allowed time.

8. While debugging you should add printing to your Prolog code so you can get a log of what it happening.

9. If you add a trace goal into your code for `guess` or `updateState` then you can see the calls made to your code from the test suite, and you can use debugging to step through the computation.

Note Well:

This project is part of your final assessment, so cheating is form of material exchange between teams, whether writ other medium, is considered cheating, and so is the sonic tronic newsgroups. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties. If you have questions regarding these rules, please ask the lecturer.