UIC

CS 511: Artificial Intelligence II
Project 2: Model-Based Reflex Agent

`Agent Theseus511`

Spring 2025

1 Concept

This projects implements a **model-based reflex agent (MRA)** that operates in the wumpus world. Specifically, it implements the agent function

$$f_{\text{MRA}} : \Omega^* \to A$$

that maps the observed sequence of percepts to an action. The MRA maintains a world model that represents the agent's state of knowledge about the world, which is used in conjunction with the observation at every time step to compute the action according to condition-action rules. The world model is updated according to the action that is executed and the observation.

2 Getting Started

The model-based reflex agent is implemented in Scala. The implementation is contained in the files `AgentFunctionImpl.scala` and `ModelBasedReflexAgent.scala`, and the code is well-documented for reference. The project directory contains a Makefile that automates building and running the project. The Makefile runs the project with the options `nonDeterministicMode` and `randomAgentLoc` set to false. The Makefile contains a `check` target that checks the system for the necessary tools (`scala`, `java`). It is recommended that the project is run after checking for the necessary tools as:-

```
$ make check
$ make #or "make run"
```

The project was tested using:-

- **Scala Version:** 3.6.3
- **Java Version:** OpenJDK 22.0.1

3 Design

The key component of the design of the MRA is the world model. The primary purpose of the world model is to represent two pieces of knowledge:-

1. how the world changes as a result of the agent's actions,
2. what the observations tell the agent about the world.

In reality, there is a third piece of knowledge: how the world evolves independent of the agent. This piece is missing in the wumpus world since it is known *a priori* that the world doesn't evolve independent of the agent. Hence, the agent's design consists of the world model, how the actions and observations update the world model, and the condition-action rules.

3.1 Definitions

Some terms that are used in describing the design of the MRA:-

- **Position :=** An ordered pair (x, y) in the grid. E.g.:- $(1, 1)$, $(2, 3)$, $(4, 1)$, etc.
- **Direction :=** North $\uparrow$, South $\downarrow$, East $\rightarrow$, or West $\leftarrow$.
- **Orientation :=** position + direction. E.g.:- $(1, 1) \rightarrow$, $(2, 3) \downarrow$, $(4, 1) \leftarrow$, etc.
- **Neighbor** (of a square) **:=** A square adjacent to the given square but not diagonally adjacent. If a square satisfying the adjacency condition is known to be unsafe with 100% certainty then it is not considered a neighbor. E.g.:- a square known to contain a pit satisfying the adjacency condition is not considered a neighbor.
- **Pit position :=** Position of a square that contains a pit.
- **Pit probability** (of a square) **:=** Probability that a given square contains a pit.
- **Pit combination :=** An ordered pair of positions (p_1, p_2), where both are possible pit positions for the two different pits. Since there are exactly two pits in the world, the complete information about the pits can be represented as a 2-tuple of positions, i.e. as a pit combination.
- **Give up :=** When the agent "gives up", all future actions are `NO_OP`s.

The agent's weapon is chosen (more like assumed) to be a bow and arrow.

3.2 The World Model and Updates

The main components of the world model that form the agent's knowledge about the world are:-

1. `agentPosition`: keeps track of the agent's position in the world. The position is represented as an ordered pair (x, y), which points to one square in the grid. It is known that the agent starts in $(1, 1)$. The agent's position is updated at every time step depending on the action. It is updated when the agent executes a `GO_FORWARD` action. It is assigned the agent's new position.
2. `agentDirection`: keeps track of the direction in which the agent is facing. This can either be North, South, East, or West. It is known that the agent starts facing east. The agent's direction is updated at every time step depending on the action. It is updated when the agent executes a `TURN_LEFT` or a `TURN_RIGHT`. It is assigned the agent's new direction.
3. `hasArrow`: indicates whether the agent has not shot, i.e. whether the agent has the "arrow". This can either be true or false. It is true until the agent uses the action `SHOOT` for the first time, after which it is set to false.
4. `numFoundPits`: keeps track of the number of pits whose positions in the grid have been pinpointed. It can either be 0, 1, or 2 since it is known that there are exactly two pits in the world. It starts out with a value of 0 since the pit positions are not known *a priori*. It is incremented when a square is flagged as a pit (detailed procedure in condition-action rules). As it turns out, there is a chance of a false positive, i.e. a square that is not a pit might be flagged as one (this chance is small). `numFoundPits` is incremented irrespective.
5. `safeSquares`: keeps track of which squares in the world are safe, i.e. contain neither a pit nor the wumpus with 100% certainty. It is a set of positions. It is updated with the neighbors of `agentPosition` if there are no observations (no `bump`, no `breeze`, no `stench`, ...); or irrespective of the observation if the wumpus is found/dead, and the pits are pinpointed.
6. `unsafeSquares`: keeps track of which squares in the world are unsafe, i.e. contain either a pit or a wumpus with 100% certainty. It is a set of positions along with a tag of *Wumpus* or *Pit*. A square that contains both the wumpus and a pit is stored twice with different tags. It is updated with the wumpus' position if it is found (detailed procedure in condition-action rules), or the position of the pits

if and when they are pinpointed. The wumpus' position and the *Wumpus* tag are added as a 2-tuple, similarly, a pit position and the *Pit* tag are added as a 2-tuple.

7. `exploredOrientationCounts`: keeps track of how many times the agent has explored different orientations. It is a mapping from orientations to counts. It is updated at every time step based on the `agentPosition` and `agentDirection`. If the 2-tuple (`agentPosition`, `agentDirection`) is already a key in the map, then its associated value is incremented, otherwise the 2-tuple is added as a key with an associated value of 1.

8. `stenchSquares`: keeps track of the squares in which a `stench` was observed. It is a set of positions. When the agent observes a `stench`, the `agentPosition` is added to `stenchSquares`.

9. `breezeSquares`: keeps track of the squares in which a `breeze` was observed. It is a set of positions. When the agent observes a `breeze`, the `agentPosition` is added to `breezeSquares`.

10. `pitCombinations`: keeps track of all possible pit combinations. It is updated using the following routine every time a `breeze` is observed:-

    1. All candidate pit combinations are computed as all 2-combinations of the set `allSquares`−`safeSquares`, where `allSquares` is the set of all (16) squares and "−" denotes the set difference.
    2. The candidate pit combinations are filtered based on the condition that the union of the sets of neighbors of both the squares in a combination must form an **exhaustive set cover** over `breezeSquares`.
    3. If there is exactly one pit combination left after filtering, then both those squares are pits with 100% certainty.
    4. Else if there is one square common among all combinations (i.e. the intersection of all combinations is a singleton set), then that is a pit with 100% certainty.
    5. Else, we have managed to shrink `pitCombinations` since we had more breeze squares to cover and possibly more safe squares to eliminate.

11. `GIVE_UP_POINT`: if the agent reaches an orientation that has already been reached this many times, then it gives up. The agent's rationale is that the gold is unreachable since it reached the same orientation so many times despite maximizing exploration (the agent is designed to prioritize less explored squares while choosing where to go). This is akin to a photographic draw in chess. The `GIVE_UP_POINT` is a constant and does not change. It must be small enough for the score to not decrease too much because of excess movement, and it must be large enough for it to be likely that the gold is unreachable if the same orientation is reached more than this many times. A value of 3 is chosen to achieve this balance.

12. `givenUp`: indicates whether the agent has given up. It is initialized to false. It is set to true when a `breeze` is observed in the starting square (hence the agent does not take chances in this case), or when the gold is unreachable for sure (i.e. when both pits and the wumpus have been found and 13 squares have been explored, or both pits have been found and 14 squares have been explored, but the gold hasn't been found), or the `GIVE_UP_POINT` has been reached.

The model updates when one or more of its component variables are updated as described. As aforementioned, since the world does not evolve independent of the agent, the world model is updated as a result of the agent's actions and observations.

3.3 Condition-Action Rules

There are 32 possible percepts and the implementation condenses these cases to 6 distinct ones. The wildcard "`_`" is used to denote a "catch-all":-

3.3.1 Case `<_,glitter,_,_,_>`

`GRAB` deterministically and win!

3.3.2 Case <_,none,none,stench,_>

if `unsafeSquares` contains a position tagged *Wumpus* **then**

1. go to the neighbor that has been explored the least number of times (hence maximize exploration)

else if `hasArrow` **then**

1. compute possible positions for the wumpus as the intersection of the sets of neighbors of all `stenchSquares`
2. **if** exactly 1 possible wumpus position is found **then**
    1. the wumpus is there with 100% certainty $\implies$ `SHOOT` (turn and `SHOOT` if necessary)
3. **else if** exactly 2 possible wumpus positions are found **then**
    1. randomly choose one of the two positions to `SHOOT` at (since if the agent misses, the wumpus is in the *other* square with 100% certainty, else it has just been shot)
4. **else** *(* the only other case is that exactly 3 possible positions are found *)*
    1. don't take chances $\implies$ turn back and go (since that is definitely a safe square)

else *(* the agent doesn't have the arrow and observes a stench *)*

1. since the agent doesn't have the arrow and observes a stench, the agent shot and missed on the last action (as per the previous condition); this is only possible when exactly 2 possible wumpus positions were computed; hence, recompute the possible wumpus positions and eliminate the one in front (since the agent shot in front by default)
2. add the other remaining position to `unsafeSquares` with the *Wumpus* tag

3.3.3 Case <_,none,breeze,stench,_>

if this is the starting square **then**

1. give up and `NO_OP`

else if `unsafeSquares` contains a position tagged *Wumpus* **then**

1. **if** `numFoundPits` is equal to 2 **then**
    1. go to the neighbor that has been explored the least number of times (hence maximize exploration)
2. **else if** a pit has been found at $(2, 2)$ and the only squares explored are $(1, 1)$, $(1, 2)$ and $(2, 1)$ **then**
    1. simply `GO_FORWARD` (more on this in design tradeoffs)
3. **else** *(* pitCombinations is updated here *)*
    1. compute pit probabilities for each neighbor as the fraction: number of times the neighbor appears in `pitCombinations` upon the size of `pitCombinations`
    2. **if** there is a unique maximum pit probability amongst the neighbors' probabilities and another nonzero probability **then**
        1. add the neighbor with the max pit probability to `unsafeSquares` with the *Pit* tag
    3. filter out the neighbors that don't have 0 pit probability
    4. out of those remaining, go to the neighbor that has been explored the least number of times (hence maximize exploration)

else if `hasArrow` **then**

1. compute possible positions for the wumpus as the intersection of the sets of neighbors of all `stenchSquares`
2. **if** exactly 1 possible wumpus position is found **then**
    1. the wumpus is there with 100% certainty $\implies$ `SHOOT` (turn and `SHOOT` if necessary)
3. **else if** exactly 2 possible wumpus positions are found **then**
    1. randomly choose one of the two positions to `SHOOT` at (since if the agent misses, the wumpus is in the *other* square with 100% certainty, else it has just been shot)
4. **else** *(* the only other case is that exactly 3 possible positions are found *)*
    1. don't take chances $\implies$ turn back and go (since that is definitely a safe square)

else *(* the agent doesn't have the arrow and observes a stench *)*

1. since the agent doesn't have the arrow and observes a stench, the agent shot and missed on the last action (as per the previous condition); this is only possible when exactly 2 possible wumpus positions were computed; hence, recompute the possible wumpus positions and eliminate the one in front (since the agent shot in front by default)
2. add the other remaining position to `unsafeSquares` with the *Wumpus* tag

3.3.4 Case <_,none,none,none,scream>

The wumpus was shot on the last action $\implies$ the square in front has become safe from unsafe $\implies$ `GO_FORWARD`.

3.3.5 Case <_,none,breeze,none,_>

if this is the starting square **then**

1. give up and `NO_OP`

else if `numFoundPits` is equal to 2 **then**

1. go to the neighbor that has been explored the least number of times (hence maximize exploration)

else if a pit has been found at $(2, 2)$ and the only squares explored are $(1, 1)$, $(1, 2)$ and $(2, 1)$ **then**

1. simply `GO_FORWARD` (more on this in design tradeoffs)

else *(* pitCombinations is updated here *)*

1. compute pit probabilities for each neighbor as the fraction: number of times the neighbor appears in `pitCombinations` upon the size of `pitCombinations`
2. **if** there is a unique maximum pit probability amongst the neighbors' probabilities and another nonzero probability **then**
    1. add the neighbor with the max pit probability to `unsafeSquares` with the *Pit* tag
3. filter out the neighbors that don't have 0 pit probability
4. out of those remaining, go to the neighbor that has been explored the least number of times (hence maximize exploration)

3.3.6 Case <_,none,none,none,none>

The agent will provably never observe a `bump`. Simply go to the neighbor that has been explored the least number of times (hence maximize exploration).

3.4 Design Tradeoffs

3.4.1 Flag pits only when 100% certain vs. with some uncertainty

In the world model, pits are flagged with 100% certainty when they are found during the update of `pitCombinations`. According to the condition-action rules, pits are flagged with some uncertainty when a square is found whose neighbors' probabilities contain a unique max probability and another nonzero probability. An example to understand the latter case better is that suppose the following probabilities are computed for the neighbors of a square: 0%, 0%, 14%, 86%. This list of probabilities contains a unique maximum, 86%, and another nonzero probability, 14%. Hence, the neighbor with pit probability 86% will get flagged as a pit although there being only an 86% chance. The decision comes down to keeping the uncertainty flagging procedure or not. It is necessary to note that flagging with uncertainty can result in false positives i.e. safe squares getting flagged as pits, which would eventually lead to death. However, the knowledge of where the pits are is generally optimal for the agent to find the gold such as in cases where it would otherwise eventually give up thinking that the gold is unreachable. Such cases are characterized by the scenario that the agent is unable to decide whether a square is a pit or not and doesn't take chances since

the certain flagging procedure maintains robust control. The uncertain flagging procedure, on the other hand, allows the agent to escape robust control and gain this knowledge faster at the cost of false positives and, in turn, death because of uncertainty. It turns out that out of the total $\binom{15}{2}$ possible pit combinations, only 4 can in false positives due to this decision. Hence, keeping the uncertain flagging procedure is favorable is more cases than it is unfavorable.

The four false positive cases are:-

1. Pits at $(1,2)$ and $(2,1)$ $\implies$ false positive at $(2,2)$.
2. Pits at $(1,4)$ and $(3,2)$ $\implies$ high chances of a false positive at $(2,3)$.
3. Pits at $(2,3)$ and $(4,1)$ $\implies$ high chances of a false positive at $(3,2)$.
4. Pits at $(2,4)$ and $(4,2)$ $\implies$ high chances of a false positive at $(3,3)$.

3.4.2 Shoot straight vs. choose randomly in certain cases of a `stench`

When a `stench` is observed and exactly 2 possible wumpus positions are computed, the strategy employed by the agent is to shoot at one of the positions and kill the wumpus or conclude that the wumpus is in the *other* position if the shot misses. The design decision here is whether the agent should always shoot straight in this situation or randomly choose one of the two positions to shoot at. The guiding principle is that it is more favorable to kill the wumpus than to simply locate it. In order to take the decision, the probability that the wumpus dies when the agent chooses randomly is weighed against the probability that the wumpus dies when the agent only shoots straight.

Let S be the proposition that the wumpus is to a side of the agent and let F be the proposition that the wumpus is in front. Let β be the proposition that exactly 2 possible wumpus positions are computed, of which one is in front. Then, $\mathrm{P}(S \mid \texttt{stench}, \beta) = \mathrm{P}(F \mid \texttt{stench}, \beta) = 0.5$. Let p be the probability that the agent shoots in front, then $1-p$ is the probability that the agent turns and shoots. Let Θ be the proposition that the agent hits the wumpus. Then,

$$\mathrm{P}(\Theta \mid \texttt{stench}, \beta) = p \cdot \mathrm{P}(F \mid \texttt{stench}, \beta) + (1-p) \cdot \mathrm{P}(S \mid \texttt{stench}, \beta) = p \cdot 0.5 + (1-p) \cdot 0.5 = \mathbf{0.5}$$

This shows that $\mathrm{P}(\Theta \mid \texttt{stench}, \beta)$ doesn't depend on p. Since $p = 1$ when the agent always shoots straight and $p = 0.5$ when the agent chooses randomly, $\mathrm{P}(\Theta \mid \texttt{stench}, \beta) = 0.5$ irrespective of the design choice. However, experimentation gave a slight edge to choosing randomly, which is why the agent is implemented to choose randomly.

3.4.3 Take a risk vs. give up in case of a `breeze` at $(1,1)$

3.4.4 Take a risk vs. give up in case of a `breeze` at $(1,2)$ and $(2,1)$

4 Results

The model-based agent was run 10,000 times and an average score of **540.6** was achieved. The following are the summary statistics:-

Minimum	1st Quartile	Median	Mean	3rd Quartile	Maximum	Std. dev.	Mode
−1089	0	963	540.6	970	1000	566.2749	0