CS 511: Artificial Intelligence II
Project 3: Utility-Based Agent

`Agent Theseus511`

Spring 2025

1 Concept

This project implements a **utility-based agent** that operates in the wumpus world. Specifically, it implements the agent function

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

that maps the observed sequence of percepts to an action. The UBA *plans* ahead in time at every time step and computes the action according to the insight gained from the forward search. It executes the action, observes the percepts, and plans from the new state for the next action.

2 Getting Started

The utility-based agent is implemented in Scala. The implementation is contained in the files `src/scala/AgentFunctionImpl.scala` and `src/scala/UtilityBasedAgent.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 above commands are for a single run by default. Of course, the `run` recipe can be updated with the `-t` option for multiple trials. Since 10,000 is a common number of trials for evaluation, a separate `make` target called `tenk` is provided that runs 10,000 trials by default using:-

```
$ make tenk
```

The score for each trial and the average score is written to `wumpus_out.txt` or to the output file you specify using the `-f` option in the recipe.

The project was tested using:-

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

3 World Models

The UBA implementation uses custom models for various aspects of its environment. The driving motivation behind the design of the models is the intuitiveness in implementation and ease of representation.

3.1 Observation Model: `Percept4`

The observations are modeled using four boolean parameters for `stench`, `breeze`, `glitter`, and `scream` respectively. The `bump` percepts are provably never observed and are promptly ignored in the implementation. This model is referred to as the **`Percept4`** model, and it is what becomes a part of the agent's planning.

3.2 Action Model: `Move`

A **move** is a **compound action** in the wumpus world. The UBA models nine moves in total:- `GoForward`, `GoLeft`, `GoRight`, `GoBack`, `Shoot`, `ShootLeft`, `ShootRight`, `Grab`, `NoOp`. For instance, the `GoRight` move is a compound action whose corresponding atomic action sequence is `TURN_RIGHT`, `GO_FORWARD`. Making the action granularity coarser helps the agent look further into the future while not wasting valuable compute on intermediate trivialities such as turns. The agent plans in terms of moves and Percept4 observations.

3.3 Representation model: `State`

The agent models a representation of the wumpus world using the variables that characterize a snapshot of the world at any point in time. These variables are the agent's position, the agent's orientation, whether the agent has the arrow, the wumpus' position, the gold's position, and the pits' positions. The variable whether the wumpus is dead or alive is implicit in the implemented model. A `state` assumes that values for all variables are known since it models a snapshot representation and does not capture uncertainty.

3.4 Uncertainty model: `BeliefState`

The UBA models its uncertainty about the world as a **belief state**. A belief state encapsulates a combination of the observable as well as unobservable variables. The observables include the agent's position, the agent's orientation, and whether the agent has the arrow. The unobservables include the position of the wumpus, the position of the gold, and the positions of the pits. The variable whether the wumpus is dead or alive is implicit. A belief state maintains a set of **belief particles**, each of which is a combination of the possible unobservables. The design choice to maintain a *set* of belief particles rather than a full-fledged probability map is a deliberate one since the belief prior is always uniform.

4 Partially Observable Monte-Carlo Planning

The UBA uses partially observable Monte-Carlo planning (POMCP) as its planning/search algorithm of choice. The algorithm combines a Monte-Carlo update of the agent's belief state with a Monte-Carlo tree search (MCTS) from the current belief state. The key idea is to evaluate each state in a search tree by the average outcome of simulations from that state. MCTS provides several major advantages over traditional search methods. It is a highly selective, best-first search that quickly focuses on the most promising regions of the search space. It breaks the curse of dimensionality by sampling state transitions instead of considering all possible state transitions.[1] It uses random simulations to estimate the potential for long-term reward, so that it plans over large horizons, and is often effective without any search heuristics or prior domain knowledge.[2] Partially observable Monte-Carlo planning (POMCP) is an extension of MCTS to partially observable environments like the wumpus world.

In the case of the wumpus world, a node in the Monte-Carlo search tree corresponds to a belief state, b, along with the number of times it has been visited, $N(b)$, and its estimated value, $Q(b)$. Hence, the root of the search tree corresponds to the current belief state. A node is expanded to obtain a child when a transition of its belief state occurs. The transition can occur via a move or a Percept4 observation. The child then corresponds to the successor belief state obtained from the transition. Hence, the search tree alternates between a layer of **action nodes** (nodes reached via actions) and a layer of **observation nodes** (nodes

[1] D. Silver and J. Veness, "Monte-Carlo planning in large POMDPs," *Advances in Neural Information Processing Systems*, vol. 23, 2010.

[2] L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. In 15th European Conference on Machine Learning, pages 282293, 2006.

reached via observations). The first node in an empty tree is an observation node, since the agent observes percepts *before* executing any action. The agent deliberates and takes decisions at the action nodes, and the observation nodes are generated by the algorithm.

With the tree structure in mind, the algorithm iteratively repeats a sequence of four steps:-

1. **SIMULATION:** Starting at the root of the search tree, moves and observations are selected, leading to successor nodes, and that process is repeated, moving down the tree to a leaf.
2. **EXPANSION:** The search tree is grown by generating new children of the selected node. Each child is obtained by a state transition via one of the possible moves.
3. **ROLLOUT:** A rollout is performed at the leaf node starting from a fully observable state sampled from the belief state at that leaf node. The moves in the rollout are chosen according to a **rollout policy** and the observations are generated by a black-box wumpus world simulator, $\mathcal{G}$. Given a fully observable state, s, and a move, m, the black-box simulator generates the successor state, the observation, and the immediate reward of executing move m in state s.

   $$(s', o, r) \leftarrow \mathcal{G}(s, m)$$

   The moves of the rollout are not recorded in the tree, and the value of the node is set to the estimated value of the rollout.
4. **BACK-PROPAGATION:** The result of the rollout is used to update all the search tree nodes going up to the root. The N-value for all the nodes is incremented, and the Q-value is updated by computing the new mean that incorporates the new incoming value.

These four steps are repeated a set number of times and a move to be returned is chosen using the selection policy (the one from the simulation step) at the root. The tree is *pruned* after every action and observation in the world to keep only the relevant branches.

The general POMCP algorithm looks like this:-

Algorithm 1 Partially Observable Monte-Carlo Planning

procedure SEARCH(h)
  repeat
    if $h = empty$ **then**
      $s \sim \mathcal{I}$
    else
      $s \sim B(h)$
    end if
    SIMULATE($s, h, 0$)
  until TIMEOUT()
  return $\underset{b}{\text{argmax}}\ V(hb)$
end procedure

procedure ROLLOUT($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  $a \sim \pi_{rollout}(h, \cdot)$
  $(s', o, r) \sim \mathcal{G}(s, a)$
  return r $+ \gamma$.ROLLOUT($s', hao, depth{+}1$)
end procedure

procedure SIMULATE($s, h, depth$)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  if $h \notin T$ **then**
    for all $a \in \mathcal{A}$ **do**
      $T(ha) \leftarrow (N_{init}(ha), V_{init}(ha), \emptyset)$
    end for
    return ROLLOUT($s, h, depth$)
  end if
  $a \leftarrow \underset{b}{\text{argmax}}\ V(hb) + c\sqrt{\frac{\log N(h)}{N(hb)}}$
  $(s', o, r) \sim \mathcal{G}(s, a)$
  $R \leftarrow r + \gamma$.SIMULATE($s', hao, depth + 1$)
  $B(h) \leftarrow B(h) \cup \{s\}$
  $N(h) \leftarrow N(h) + 1$
  $N(ha) \leftarrow N(ha) + 1$
  $V(ha) \leftarrow V(ha) + \frac{R - V(ha)}{N(ha)}$
  return R
end procedure

4.1 Parameters

The forward search is characterized by parameters that have a direct effect on the quality of the search and hence on the performance of the agent. These parameters are:-

- The **time horizon**, H, is the maximum depth to which a simulation (in the simulation step) or a rollout (in the rollout step) is carried out. Since the simulations and rollouts are carried out on fully observable states sampled from the belief states, the estimates from the search are not completely accurate. Higher values of H "emphasize" this inaccuracy and end up performing worse. In rollouts with high H, the agent almost always ends up dying, and the best action propagated is always a `NO_OP`. Basically, the agent performs severely suboptimally. In the UBA implementation, $H = 15$.
- The **discount factor**, $\gamma \in [0, 1]$, is the weight associated with future value as compared to the current value. $\gamma = 1$ implies that future values are weighed the same as the immediate or current values. For instance, if there is a higher chance of getting the gold through a pit square, the agent will *voluntarily waltz into the pit square*, disregarding the immediate danger ahead. Experimentally, lower values of γ seemed to perform better for the wumpus environment. In the UBA implementation, $\gamma = 0.2$. With this value, the agent weighs immediate threats high enough that it never dies by waltzing into a pit or a wumpus. The value estimate for executing a move m in state s at timestep t is calculated as:-

  $$U_t(s, m) = R(s, m, s') + \gamma U_{t+1}(s', m')$$

  where R is the reward for executing move m in state s. The value is calculated like this both for simulations and rollouts. Ultimately, the value of a node is the mean of the values of its children. These means are updated in the back-propagation step.
- The **discount horizon**, ϵ, dictates the depth where onwards the discount factor effectively becomes 0. At a depth d, the value is discounted by a factor of γ^d. The discount horizon says that if $\gamma^d < \epsilon$, then stop the search and disregard future values. Just as the time horizon bounds the depth d, the discount horizon bounds the effective discount γ^d. In the UBA implementation $\epsilon = 0$.
- The **number of simulations**, T, is arguably the most important determiner of the optimality agent function. It is a property of MCTS and, by extension, of POMCP that as T approaches infinity, the agent function becomes more optimal. The UBA implementation has $T = 1000$. This itself took 29.75 hours for evaluation using 10,000 runs!

4.2 Policies

There are two key policies guiding the search: the rollout policy, π_r, and the selection policy, π_s.

4.2.1 Rollout policy

The rollout policy samples a move given the belief state, $m \sim \pi_r(b)$. Each belief state is the result of a **history** i.e. an alternating sequence of actions and observations. In the POMCP paper, almost everything is in terms of histories rather than belief states since the belief states might be large. In the wumpus world, however, the size is manageable. The histories are of significance in looking up the most recent observation, which conditionally determines the action in the rollout policy. Let the proposition $\omega(p)$ be true if percept p is observed in the most recent observation. In the UBA implementation, the rollout policy for a belief state b, $\pi_r(b)$:-

if ω(`glitter`) **then** `Grab`
else if ω(`stench`) and the agent has the arrow **then** $m \sim$ Uniform({`Shoot`, `ShootRight`, `ShootLeft`})
else *(* favor exploration *)* $m \sim$ Uniform({`GoForward`, `GoLeft`, `GoRight`, `NoOp`})

The rollout policy is responsible for determining the quality of rollouts, which in turn, affects the quality of the search. Move ordering is a way to improve the quality of rollouts, which is used in the project. Move ordering says that depending on the situation, some moves are preferred according to some condition to be the outcome of π_r.

4.2.2 Selection policy

The selection policy chooses the action to take at an observation node. If the policy focuses only on *exploitation*, that is choosing the action that leads to the action node with the highest value, it is suboptimal since there must also be an element of *exploration*. Hence, the selection policy first tries to explore the action nodes that are leaves (unexplored). It does so by falling back on the rollout policy. If the rollout policy in the current belief state returns an action that leads to a leaf node, that branch is taken. Else, if there are other unexplored nodes, then one of them is chosen randomly only if the corresponding action is not a shooting action. If there are no unexplored action children, or the only ones are led to by a shooting action, then the selection policy uses the **upper-confidence tree (UCT) bound** using the UCB1 formula:-

$$\pi_s(n) = \arg\max_m \; w_1(m)Q(n) + w_2(m)C\sqrt{\frac{\ln[N(\texttt{PARENT}(n))]}{N(n)+1}}$$

The first addend term is the exploitation term and the second one is the exploration term. C is the exploration constant, which has a value of $\sqrt{2}$. The terms are weighted by weights w_1 and w_2, which are the exploitation and exploration weights respectively. The weights are a function of the move so that the exploration and exploitation can be weighed differently for different moves. For instance, the agent wants to explore the option of shooting only when it has observed a `stench` to increase the chance of hitting the wumpus. Hence, the exploration weight for shooting moves increases linearly with the number of times a `stench` is observed in the history. The exploration weight is initially 0 so that the agent chooses to shoot only if it thinks the estimated value is high enough. All exploitation weights are set to 1 in the implementation since exploitation weights overestimate negative values, and the agent is already risk-averse. A further overestimation of danger would cause it to simply give up all the time, which is suboptimal.

4.2.3 Reward function

The value of a reward for a move is the sum of the rewards for all actions in the corresponding action sequence as defined by the rules of the wumpus world. Additionally, the agent tacks on a *heuristic* which draws the agent towards favorable states and consists of two parts: a *gold score* and a *wumpus score*.

- The *gold score* penalizes actions that increase the Manhattan distance between the gold and the agent. Recall that the simulations and rollouts involve fully observable states sampled from belief states. Hence, the positions of the agent and the gold are perfectly known. The gold score heuristic is expressed as a **weighted potential function**:-

  $$h_1(s, m, s') = w(\phi(s') - \phi(s))$$

  The **shapeness theorem** guarantees that tacking on this term does not change the utility preferences. In the UBA implementation, the potential function ϕ is nothing but the Manhattan distance between the agent and the gold. Since lower Manhattan distances are better, the weight $w < 0$, specifically, $w = -4$ in the implementation.

- The *wumpus score* favors states that don't have a wumpus (i.e. the wumpus is dead) over states that do.

  $$h_2(s, m, s') = \begin{cases} 0 & \text{if } s' \text{has the wumpus} \\ 9 & \text{if } s' \text{does not have the wumpus} \end{cases}$$

  The value 9 is chosen to achieve the balance between wasting the shot unnecessarily and prioritizing shooting actions when a `stench` is observed.

5 Results

The utility-based agent was run 10,000 times and an average score of **513.1224** was achieved. The following are the summary statistics:-

Minimum	1st Quartile	Median	Mean	3rd Quartile	Maximum	Std. dev.	Mode
−1013	−16	961	513.1224	987	1000	505.7418	1000

6 Limitations

- The agent **almost never dies** (!) since it weighs immediate dangers like waltzing into a pit or the wumpus heavily. However, in cases where it is "stuck" because of perception of danger, for instance because of a `breeze` in $(1, 1)$, the agent acts suboptimally by arbitrarily shooting and grabbing at some time steps instead of optimally completely giving up.
- In cases where the gold is unreachable, the agent keeps randomly moving about and decreasing the score instead of giving up after a point. Perhaps the exploration weights of the movement actions should decrease as a function of time and the exploration weight of a `NoOp` should increase. These functions should be well thought out.
- The agent lacks rigorous insight for the search parameters and the policies. The histories' knowledge could have been much better utilized in designing the policies and complex exploration weight functions that perform optimally.

7 References

1. The POMCP paper:- D. Silver and J. Veness, "Monte-Carlo planning in large POMDPs," *Advances in Neural Information Processing Systems*, vol. 23, 2010.
2. An example Python implementation of POMCP.

8 Reflection

- I should've described the wumpus world using the formal definition of a POMDP, and incorporated that into the deisgn description. As compensation, I'm reiterating the formal definition of a POMDP here:

  A *POMDP* is a 7-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \gamma \rangle$, where

  - $\mathcal{S}$ is a finite set of states,
  - $\mathcal{A}$ is a finite set of actions,
  - $\mathcal{O}$ is a finite set of observations,
  - $\mathcal{P}$ is a state transition probability matrix,

    $$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s,\ A_t = a]$$

  - $\mathcal{R}$ is a reward function,

    $$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} \mid S_t = s,\ A_t = a]$$

  - $\mathcal{Z}$ is an observation function,

    $$\mathcal{Z}^a_{s'o} = \mathbb{P}[O_{t+1} = o \mid S_{t+1} = s',\ A_t = a]$$

  - γ is a discount factor, $\gamma \in [0, 1]$.

- I fixed the `expandFrom` method so that if the new belief state is already contained in the tree then a new node for it is not created. The existing node is pointed to instead.