CS3243 HW2T Agent

Below are the algorithms that make up the AI for our agent. We chose to use a model-based agent, as a purely reflex based agent would not take advantage of the possibilities of logical inference in the world, and the lack of information about the proximity of the gold would have made a goal-based agent useless.

The agent explores the map by visiting all the safe, unexplored neighbours of its current tile (which we call base), updating its model, and then selecting a safe neighbour to be the next base. This continues until the agent visits a tile containing the gold, whereupon it picks up the gold and then tries to exit the world as quickly as possible.

If the agent ever finds itself in a position in which it cannot move to a safe, unexplored tile, it will return to the origin and start over. This eliminates the risk of our agent trapping itself in a region of the map and exiting prematurely. However, this also means that our agent refuses to accept the possibility that there is no way of reaching the gold (if any exists). In other words, our AI will *never give up*.

We ensure that we only move to tiles we have some knowledge about by making cross-moves before diagonal ones. As we also shuffle the order in which we make diagonal moves, our choice of the next base (which is the tile reached via the last executed safe move) has a degree of randomness. This should reduce the risk of our agent boxing itself into a region (and even if it does, as explained above, we will just start over).

List of Algorithms

Algorithm 1 Search Function - **explore**()

Require: CROSS MOVES ← [(1,0),(-1,0),(0,1),(0,-1)]
Require: DIAGONAL MOVES ← [(1,1),(-1,-1),(1,-1),(-1,1)]

```
1: KnowledgeBase ← ∅
2: explored ← ∅
3:
4: origin ← (0,0)
5: origin state ← perceive
6: KnowledgeBase.put(origin state)
7: infer(origin)
8: explored(origin.X,origin.Y) ← true
9: base ← origin
10:
11: loop
12:   MOVES ← CROSS MOVES ∪ shuffle(DIAGONAL MOVES)
13:   for all Move m in MOVES do
14:     newX ← base.x + m.x
15:     newY ← base.y + m.y
16:     movesMade ← 0
17:     if isSafe(newX,newY) ∧ ¬ explored(newX,newY) then
18:       moveTo(newX,newY)
19:       movesMade ← movesMade + 1
20:       state ← perceive
21:       KnowledgeBase.put(state)
22:       infer(state)
23:       explored(newX,newY) ← true
24:       if isGlittery(state) then
25:         pickUpGold
26:         foundTheGold ← true
27:         escape(newX,newY)
28:       end if
29:       if ¬isBlack(state) then
30:         nextBase ← (newX,newY)
31:       end if
32:       moveTo(base.x,base.y)
33:     end if
34:   end for
35:   if movesMade = 0 then
36:     escape(base.x,base.y)
37:   end if
38:   base ← nextBase
39:   moveTo(base.x,base.y)
40: end loop
```

Algorithm 2 Inference Function - **infer**()

Require: a state to infer knowledge about
Require: CROSS MOVES ← [(1,0),(-1,0),(0,1),(0,-1)]
Require: a KnowledgeBase to update

```
1:  for all Move m in CROSS MOVES do
2:    adjacentX ← state.x + m.x
3:    adjacentY ← state.y + m.y
4:    if KnowledgeBase.contains(adjacentX,adjacentY) then
5:      adjState ← KnowledgeBase.get(adjacentX,adjacentY)
6:    else
7:      adjState ← new State
8:    end if
9:
10:   if isEmpty(state) then
11:     adjState.isEmpty ← true
12:   else
13:     if isBreezy(state) then
14:       adjState.pitPossibility ← adjState.pitPossibility + 1
15:     end if
16:     if isSmelly(state) then
17:       adjState.wumpusPossibility ← adjState.wumpusPossibility + 1
18:     end if
19:   end if
20:   KnowledgeBase.update(adjState)
21: end for
```

Algorithm 3 Safety Evaluation Function - **isSafe**()

Require: a position (x,y) to evaluate
Require: a KnowledgeBase
1: state ← KnowledgeBase.**get**(x,y)
2: **if isBlack**(state) **then**
3: **return** $false$
4: **end if**
5: **if isEmpty**(state) ∨ (state.pitPossibility = 0 ∧ state.wumpusPossibility = 0) **then**
6: **return** $true$
7: **else**
8: **return** $false$
9: **end if**

Algorithm 4 Escaping Function - **escape**()

Require: a KnowledgeBase
Require: a startingPosition
Require: whether we foundTheGold
1: currentPosition ← startingPosition
2: **repeat**
3: nextPosition ← a safe neighbour of currentPosition that minimises the straight line distance to (0,0)
4: **moveTo**(nextPosition)
5: currentPosition ← nextPosition
6: **until** currentPosition = (0,0)
7: **if** ¬foundTheGold **then**
8: **explore**
9: **end if**