Assignment 02: Online Search for Hunt the Wumpus

Group project by
Green team

Master in Computational Data Science
Artificial Intelligence: Methods and Applications

LIBERA UNIVERSITÁ DI BOLZANO
Bolzano, Alto Adige

Authors: Di Panfilo Marco, Lorefice Alessandra, Mugisha Denis, Pellè Gianluigi

Professor: Sergio Tessaris

Submission date: 16/12/2020

Academic year: 2020-2021

ABSTRACT

The task of the assignment was to implement an uninformed player to solve the "Hunt the Wumpus" game getting the best reward using an online search algorithm.

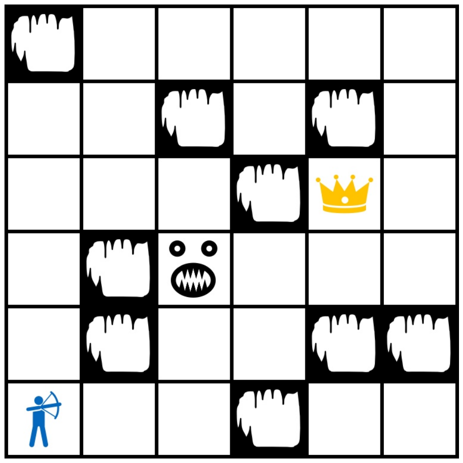
A brief description of the game is as follow: we are in a chess-like world environment where each cell can either be empty or a pit, and there might be a wumpus in it. The agent can only move in empty cells (otherwise he would die either by falling into a pit or killed by the wumpus) and the movement is restricted by the orientation the agent has before performing the movement action. The world contains exactly one gold, which has to be grabbed by the agent if that is possible, and exactly one wumpus (monster) which can be killed if necessary in order to move into its location. Since the player we had to implement was an uninformed one, it couldn't see the whole world configuration upon request, so it had to explore the world using its sensors and elaborating its percepts as it progresses in the game to get information about the environment. The agent perceives a BREEZE, if there is one or more pits in the 4 ortogonally adjacent locations (but doesn't know where exactly that pit is, and how many of them there are). It perceives a STENCH, if the wumpus is in one of the ortogonally adjacent locations, and it perceives BUMP if it bumps on the border of the world. The last percept the agent can perceive is GLITTER, which tells the agent that there is a gold in that location. The aim of the game is to get the best reward possible from trying to grab the gold and climbing out of the world.

TABLE OF CONTENTS

Chapter 1

PROBLEM DESCRIPTION

Let's have a look at the problem:



We are a hunter looking for treasures in a **grid-map world** environment. The goal of the agent is to try to grab the gold and then exit the world (from where we came from) with the **cheapest sequence of actions**.

The agent (aka the hunter) has an orientation and he can only move straight in the direction he is pointing to, otherwise, he first has to rotate (either LEFT or RIGHT) to be able to move to a different tile. The possible orientations an agent can have are North, East, South or West, so he can only **move orthogonally**. In the world there may be several pits: if the agent ever enters such a tile he will die immediately and the game will be over. The agent also has an **arrow** (just one) that he can use to kill the wumpus; if you don't kill the wumpus and you enter the tile containing it the agent will die and the game will terminate.
The available actions for the agent are:

- LEFT
- RIGHT
- MOVE
- SHOOT

- GRAB
- CLIMB

They all cost 1, except shooting costs 10 if you shoot with an arrow (shooting without an arrow is still possible but will have no effect). The same goes for the GRAB and CLIMB action, you can grab wherever you are, but if the agent is not in the gold location or the exit location respectively, then the action will have no effect. If you grab the gold, you'll get a 1000 points reward, and if you die, you'll get a -1000 points reward.

But, since we have to implement an online search, our world is not fully visible upon request, so we need to explore it to get useful information about it. Information are gathered with some sensors the agent has, and they are perceived at the end of each action performed by the agent. The agent perceives a BREEZE, if there is one or more pits in the 4 ortogonally adjacent locations (but doesn't know where exactly that pit is, and how many of them there are). It perceives a STENCH, if the wumpus is in one of the ortogonally adjacent locations, and it perceives a BUMP if it bumps on the border of the world. The last percept the agent can perceive is GLITTER, which tells the agent that there is a gold in that location.

The aim of the game is to get the best reward possible from trying to grab the gold and climbing out of the world, so we implemented a strategy - which will be later explained in details - to go and visit all the safe locations and gather the most possible information, then it tries kill the wumpus to discover new safe locations to explore, and, if it couldn't yet find the gold it decides whether to risk by exploring the most probable safe locations or go home. If the agent perceives the GLITTER while exploring safe locations, it would stop its search and go back home with the shortest route to the exit location given the safe locations he is aware of.

Chapter 2

MODEL DESCRIPTION

2.1 SmartCoordinate and SmartVector

Since the movement was restricted to the orientation of the agent, we decided to model the world as a **vector space** introducing the *linear_space* module where we defined the *SmartCoordinate* class, to represent locations in the world, and the *SmartVector* class to represent the orientation of the agent.

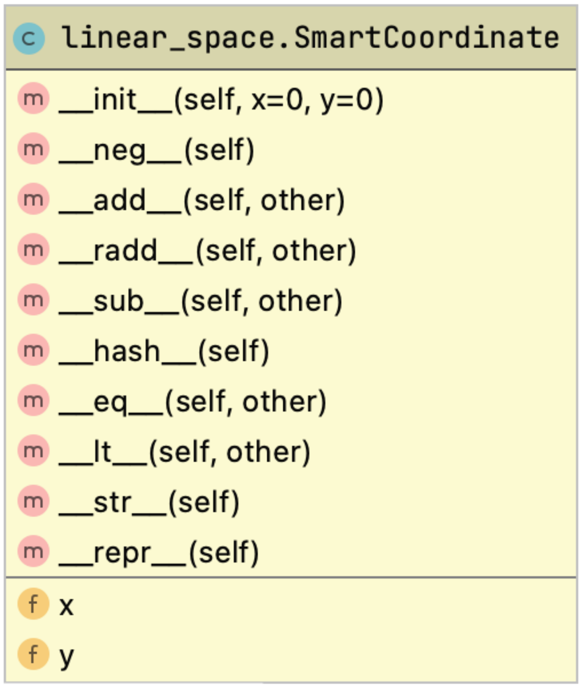
We implemented the operator (*SmartCoordinate* + *SmartVector*) so that you can sum a location with an orientation and get a *SmartCoordinate* representing the location of the tile you're moving into following that orientation.
We also defined some other helper methods. A noteworthy one is the ***get_perpendicular_vector_clockwise()*** which returns the perpendicular vector (in clockwise order) to the vector calling the method, which is useful for some future calculations.
The definition of both classes is shown in Figure 2.1.

2.2 KnowledgeBase class

Focusing on the online search, we started to develop the KnowledgeBase class to model all the information we could infer from the percepts of the agent. This class represents all the information the agent could infer from its entire exploration. It keeps track of the agent location, orientation and arrow availability, by keeping them up to date based on all the previous actions performed by the agent. It also keeps track of the world size. It starts by assuming the world size is of width 1 and height 1 and then it increases these values by 1 every time the agent moves easter or norther than the width/height size it is aware of. In the case the agent perceives a bump the knowledge base will not update the width/height value but it will flag the final world size for that dimension.

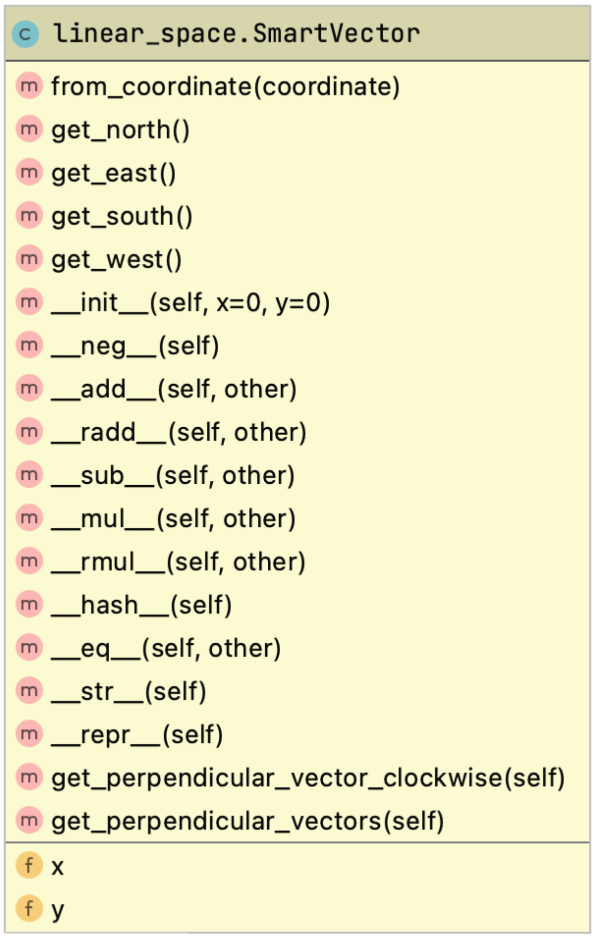
The solution we developed works both for squared and rectangular world environments, but since in this case the world for the game was assured to be a square, we added a flag *IS_SQUARE_SIZE* to improve the efficency of the code using this information. The knowledge base also contains a lot of information related to specific environment locations, the most relevant ones are the following:

- *visited_locations*: list of all locations the agent has been moved into (which do not need to be visited/explored again, since the inferred information were already added to the



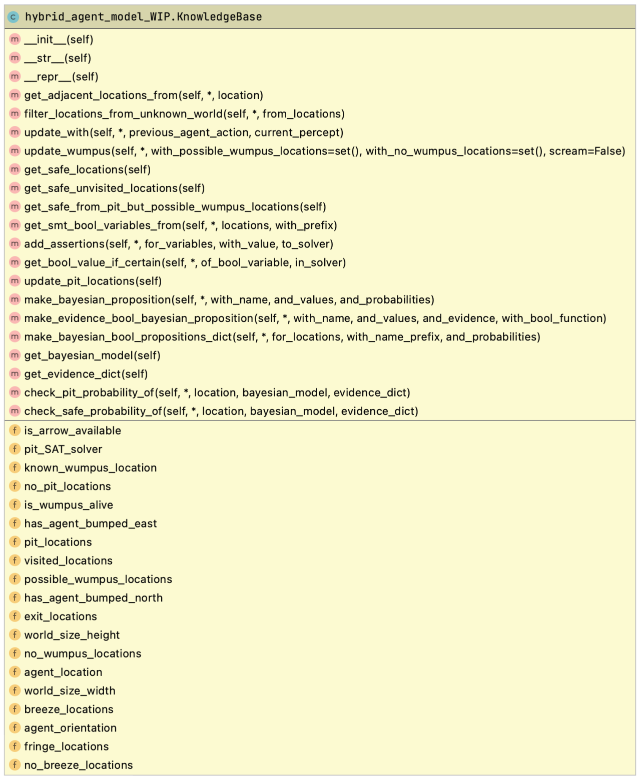
```
linear_space.SmartCoordinate
__init__(self, x=0, y=0)
__neg__(self)
__add__(self, other)
__radd__(self, other)
__sub__(self, other)
__hash__(self)
__eq__(self, other)
__lt__(self, other)
__str__(self)
__repr__(self)
x
y
```

(a) *SmartCoordinate class diagram*



```
linear_space.SmartVector
from_coordinate(coordinate)
get_north()
get_east()
get_south()
get_west()
__init__(self, x=0, y=0)
__neg__(self)
__add__(self, other)
__radd__(self, other)
__sub__(self, other)
__mul__(self, other)
__rmul__(self, other)
__hash__(self)
__eq__(self, other)
__str__(self)
__repr__(self)
get_perpendicular_vector_clockwise(self)
get_perpendicular_vectors(self)
x
y
```

(b) *SmartVector class diagram*



```
hybrid_agent_model_WIP.KnowledgeBase
m __init__(self)
m __str__(self)
m __repr__(self)
m get_adjacent_locations_from(self, *, location)
m filter_locations_from_unknown_world(self, *, from_locations)
m update_with(self, *, previous_agent_action, current_percept)
m update_wumpus(self, *, with_possible_wumpus_locations=set(), with_no_wumpus_locations=set(), scream=False)
m get_safe_locations(self)
m get_safe_unvisited_locations(self)
m get_safe_from_pit_but_possible_wumpus_locations(self)
m get_smt_bool_variables_from(self, *, locations, with_prefix)
m add_assertions(self, *, for_variables, with_value, to_solver)
m get_bool_value_if_certain(self, *, of_bool_variable, in_solver)
m update_pit_locations(self)
m make_bayesian_proposition(self, *, with_name, and_values, and_probabilities)
m make_evidence_bool_bayesian_proposition(self, *, with_name, and_values, and_evidence, with_bool_function)
m make_bayesian_bool_propositions_dict(self, *, for_locations, with_name_prefix, and_probabilities)
m get_bayesian_model(self)
m get_evidence_dict(self)
m check_pit_probability_of(self, *, location, bayesian_model, evidence_dict)
m check_safe_probability_of(self, *, location, bayesian_model, evidence_dict)
f is_arrow_available
f pit_SAT_solver
f known_wumpus_location
f no_pit_locations
f is_wumpus_alive
f has_agent_bumped_east
f pit_locations
f visited_locations
f possible_wumpus_locations
f has_agent_bumped_north
f exit_locations
f world_size_height
f no_wumpus_locations
f agent_location
f world_size_width
f breeze_locations
f agent_orientation
f fringe_locations
f no_breeze_locations
```

Figure 2.2: KnowledgeBase class diagram

knwoledge base)

- *fringe_locations*: list of all locations which are ortogonally adjacent to visited locations but have not been explored yet (the ones the agent can safely reach, but are not guaranteed to be safe)

- *no_pit_locations*: list of all locations where we could infer that there must not be a pit inside

- *no_wumpus_locations*: list of all locations where we could infer that there must not be a wumpus inside

Eventually, the knowledge base models all the information it could infer for the wumpus. KnowledgeBase does something more than just storing the information passed in from the agent, it also uses a bayesian network to elaborate all the data it has and calculates the probability of the presence of a pit in a certain location and the safe probability of a location. The definition of the class is shown in Figure 2.2.

2.3 HuntWumpusHybridAgent

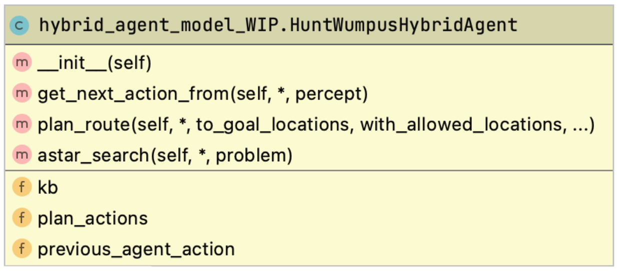
This class defines the strategy of the online search and implements a method *get_next_action_from(percept)* that gives the agent the next action to perform. The strategy we developed is the following:

1) The agent explores all the locations it can infer to be 100% safe: it looks at the fringe, selects the closest safe location and makes a plan to reach that location. If the knowledge base could infer the exact location of the wumpus, that location will be considered safe as well, but will have a higher path cost given that we need to kill the wumpus beforehand to move into that location.

2) Once there are no more safe locations, we can try shooting the wumpus (if it is still alive and the agent didn't use the arrow yet) to infer new safe locations: we check the possible wumpus locations, and given that they all have the same probability of being the correct one, we select the one with the lowest pit probability, as it doesn't make sense to shoot a location where we cannot move into because of the presence of a pit (if every possible wumpus location has a pit probaility of 70% or more, then it doesn't make sense to kill the wumpus at all, and we move straight to 3)). If we could hear the scream, then we can update the knowledge base and tell it that there is no wumpus in all *possible_wumpus_locations*, otherwise we can only tell it that the location we shot to is free from the wumpus. It may be the case that the possible wumpus locations were only 2, so now we know where the wumpus is. Anyway, we ask again the knowledge base for new safe locations, and we go and explore them.

3) If there are no more safe locations and the agent has no arrow available, then we need to decide whether to take a risk and explore a location which is not guaranteed to be safe or go back home and climb out. We use Bayesian Network to choose which of the unsafe cells is the most unlikely to be dangerous. Particularly, we use it for getting the probability of a cell to be safe (using the method *check_safe_probability_of*) or to be a pit (using the method *check_pit_probability_of*). Since the cause of danger is the presence of a wumpus or a pit and the effect is the perception, then we built the Bayesian Networks using the fact that the presence of an obstacle is a parent of the perception.
We define a 75% probability as the treshold to go and risk to explore a new location, if there are no locations with a safe probability greater than 75% than the agent go back home. As the agent explores new cells it can discover new safe cells and go back to plan 1) and then back to plan 3).

The strategy stops as soon as the agent enters a location and perceives GLITTER. In that case, a new plan is made, which is composed of the GRAB action, the sequence of actions to



hybrid_agent_model_WIP.HuntWumpusHybridAgent

```
__init__(self)
get_next_action_from(self, *, percept)
plan_route(self, *, to_goal_locations, with_allowed_locations, ...)
astar_search(self, *, problem)
kb
plan_actions
previous_agent_action
```

Figure 2.3: HybridWumpusAgent class diagram

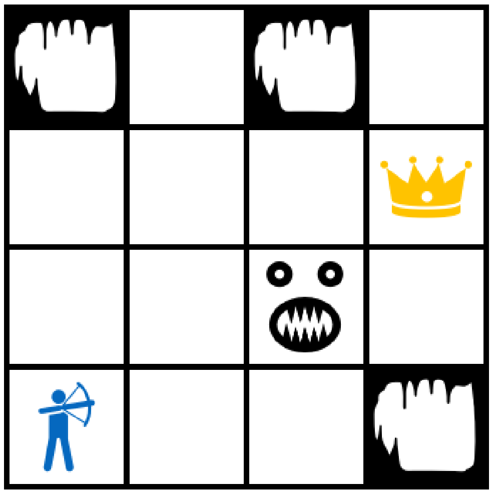


Figure 2.4: Example of Wumpus world

reach the exit, and the CLIMB out action. To calculate the sequence of actions to reach the exit an offline search is performed given the safe locations the knowledge base is aware of. The definition of the class is shown in Figure 2.3.

Strategy Examples

In the following section we are going to illustrate the strategy based on a example world in Figure 2.4. We can see that there are 3 pits on the border and the wumpus in the middle of the world. The gold is somehow hidden behind the wumpus and close to pits.

As described in point 1) of the strategy the agent will try first to explore all the safe cells as it is shown in 2.5. Meanwhile exploring the safe cells the agent can already infer the location of some pits and wumpus as shown in Figure 2.6. He can be sure that in the corner is a pit, cause the cell below it has all safe neighbours except the corner. Furthermore he can locate precisely the wumpus cause he explored 3 cells with stench next to him and, by knowing that there is only 1 wumpus, then that is the only possible location. Since there are no other safe cells left to explore the agent shoots the wumpus and kills him 2.7.

After killing the wumpus the agent will explore the last safe cell and infer that in the right-

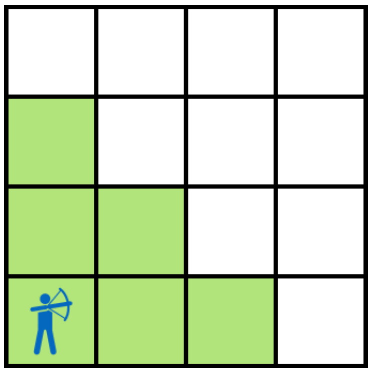


Figure 2.5: Exploration of safe cells

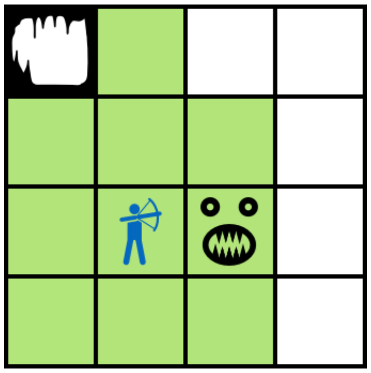


Figure 2.6: Perceiving breeze and stench

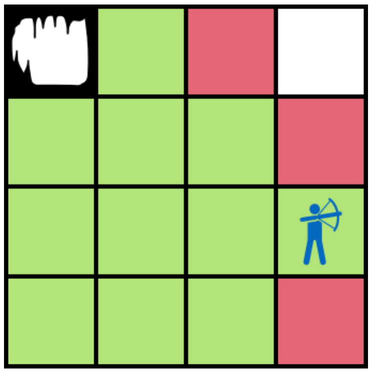


Figure 2.7: Possible pits

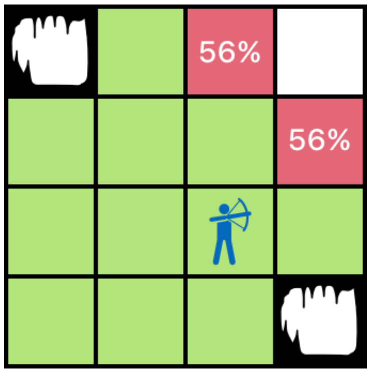


Figure 2.8: Calculation of safe probability

bottom corner is a pit. The agent can not infer for sure if the cells next to the upper right corner are pits or not. At least one of them is a pit. In order to take the decision if he should risk or not a calculation based on Bayesian Networks is performed. The result shows that given that each cell has a 0.2 probability of being a pit, those cells have a probability of 0.56 of being a pit based on the breezes the agent perceived 2.8. Since this is higher then the risk threshold, then the agent will go back to the exit location without the gold.

2.4 HuntWumpusProblem

Since we needed to compute three kind of offline search very similar from one another we decided to model a single parameterized search problem to accomodate all such needed searches.

A detailed description of the HuntWumpusProblem class is defined later on chapter 3. The definition of the class is shown in Figure 2.9. We also modeled the HuntWumpusNode and HuntWumpusState classes according to the problem definition, and they are illustred on figures 2.10 (a) and (b).

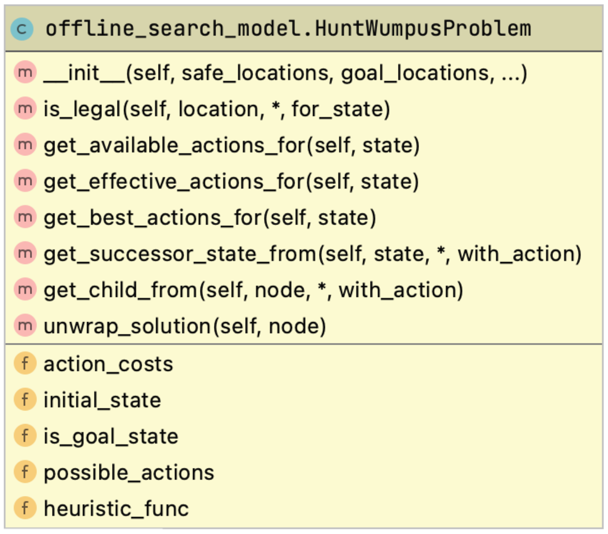
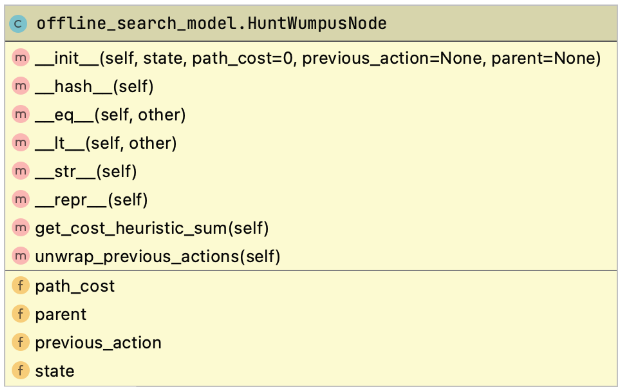
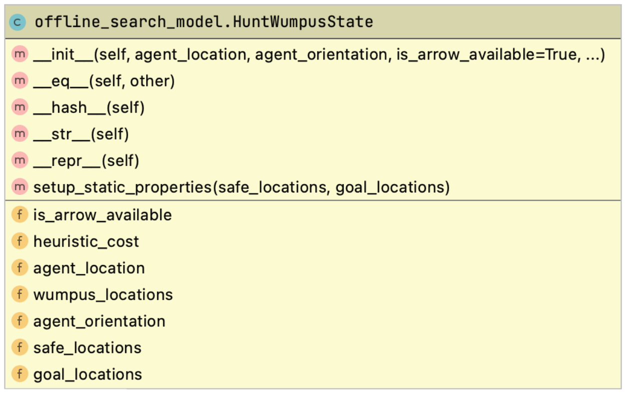

c offline_search_model.HuntWumpusProblem
m __init__(self, safe_locations, goal_locations, ...)
m is_legal(self, location, *, for_state)
m get_available_actions_for(self, state)
m get_effective_actions_for(self, state)
m get_best_actions_for(self, state)
m get_successor_state_from(self, state, *, with_action)
m get_child_from(self, node, *, with_action)
m unwrap_solution(self, node)
f action_costs
f initial_state
f is_goal_state
f possible_actions
f heuristic_func

Figure 2.9: Problem class diagram


c offline_search_model.HuntWumpusNode
m __init__(self, state, path_cost=0, previous_action=None, parent=None)
m __hash__(self)
m __eq__(self, other)
m __lt__(self, other)
m __str__(self)
m __repr__(self)
m get_cost_heuristic_sum(self)
m unwrap_previous_actions(self)
f path_cost
f parent
f previous_action
f state

(a) *HuntWumpusNode class diagram*



(b) *HuntWumpusState class diagram*

Figure 2.10

Chapter 3

SEARCH

We defined a single problem for the offline search which is general purpose and can be used to accomplish each of our needs. The problem searches for the best sequence of actions to go from the agent location to one of the goal locations passed in input, heading to the closest one. We defined the transition model for the actions MOVE, LEFT, RIGHT and SHOOT, and we defined the locations were the agent can move in as the *safe_locations*; the problem also gets in input the possible wumpus locations, in which the agent can only move if he first tries to shoot the wumpus; we tell the problem if he can try to shoot the wumpus by filtering or not the SHOOT action from the possible actions passed in input. We have 3 cases where we need to compute an offline search, which are described as follow:

1) we grabbed the gold, and we need to calculate the shortest sequence of actions to exit the game. We instantiate a problem with the following informations:

    - **safe_locations**: all locations where we are sure there is no pit, including the ones with a possible wumpus([1])
    - **goal_locations**: {*exit_location*}
    - **possible_actions**: {MOVE, LEFT, RIGHT, SHOOT}
    - **is_goal_state**: agent location is in the exit location
    - **heuristic_func**: *best_neighbour_smart_manhattan*(from: *agent_location*, to: *exit_location*)
    - **agent_location**: agent location
    - **agent_orientation**: agent orientation
    - **is_arrow_available**: if the agent has the arrow available
    - **possible_wumpus_locations**: {*possible_wumpus_locations*}

2) we are visiting the safe locations and we want a sequence of actions to reach the closest safe location we haven't visited yet. We instantiate a problem with the following informations:

    - **safe_locations**: all locations where we are sure there is no pit and no wumpus([2])

[1] note that going back we can kill a possible Wumpus location if that makes us discover a shorter path.

[2] There is an hack we adopted: if we know where exactly the wumpus is, then we consider that location to be safe from wumpus, since we can shoot it and visit it, because it can be cheaper killing the wumpus than reaching a very far away safe location. In that case we also pass in the SHOOT action in the possible actions.

- **goal_locations**: {safe locations inside the fringe locations}
- **possible_actions**: {MOVE, LEFT, RIGHT, SHOOT}
- **is_goal_state**: agent location is in a safe location inside the fringe locations
- **heuristic_func**: the minimum from the *best_neighbour_smart_manhattan* of each safe location in the fringe locations
- **agent_location**: agent location
- **agent_orientation**: agent orientation
- **is_arrow_available**: if the agent has the arrow available
- **possible_wumpus_locations**: {*possible_wumpus_locations*}

3) we already visited all safe locations, and we want to try and kill the wumpus to explore new safe locations. Given that the wumpus has the same probability of being on any of the possible wumpus locations, we first choose the one with the lowest pit probability (0.0 is preferred) and then we run the offline search to reach the wumpus location, killing it before moving into its location. In case there are more than one wumpus locations with the same lowest pit probability, then we pass to the problem all of them and the search will go and try to kill the closest one: we instantiate a problem with the following informations:

   - **safe_locations**: all locations where we are sure there is no pit and the Wumpus locations we are going to try and kill.
   - **goal_locations**: Wumpus locations we are going to try and kill
   - **possible_actions**: {MOVE, LEFT, RIGHT, SHOOT}
   - **is_goal_state**: agent location is in one of the Wumpus locations we are going to try and kill
   - **heuristic_func**: the minimum from the *best_neighbour_smart_manhattan* of each *wumpus_location* passed in
   - **agent_location**: agent location
   - **agent_orientation**: agent orientation
   - **is_arrow_available**: if the agent has the arrow available
   - **possible_wumpus_locations**: {*wumpus_locations*}

Chapter 4

SAFETY THRESHOLD

We made some tests to check the agent's performances and to tune the best risk threshold. We started with a 70% safe threshold probability and it seemed quite good, then we made 1000 to 2000 runs for different threshold values starting from 0.6 to 0.9 and on different world sizes, to see if there was a threshold looking good for all different world sizes. All results are summarized on table 4.1 and 4.2.

From the tables we can see that as expected the average outcome decreases with the size of the world. This is due to higher explorations costs and probably the agent has to risk more to reach the gold and falls more often in a pit (Table 4.2). Furthermore, we can see that the threshold influences the average outcome only slightly in small worlds. Furthermore, the results showed also a high standard deviation (not shown in table only in Appendix) such that a more detailed analysis with higher samples or categorising the data in died, gold, no gold would be helpful to understand better what the key factors are.

Based on our analysis we can say that a lower risk threshold seems to be better only for smaller worlds 4x4 and 5x5. After this the risk values 80% and 90% show the best average. From the Table 4.2 we see that the risk threshold of 80% and 90% behave the same way since, they never died and therefore most likely never risked to go in a pit. Also the overall performance for worlds (4x4 - 8x8) seems to be best for risk thresholds, therefore we choose 0.8 as our best value. An alternative solution could be to start with a lower risk value and increase it, when the world increases.

Another thing to consider is that having a low risk threshold results in a lot more time computation since the Bayesian network is being called more times, so, increasing the risk threshold gets a lower average outcome but a faster execution, whereas having a low risk threshold results in a better average outcome but a slower execution. Since high thresholds have also a very good average outcome it could be a good option to always go home when no safe cells are left. This avoids a heavier computations of Bayesian networks or SAT-solver. There was no time to do some tests to confirm this hypothesis.

threshold	4x4	5x5	6x6	7x7	8x8	average
60	344.202	234.392	102.904	-14.722	-56.109	122.1333
70	334.069	256.19	182.768	120.336	65.394	191.7515
72	351.238	276.18	193.095	126.244	47.048	198.7610
74	337.555	265.522	180.433	153.147	60.656	199.4625
76	353.359	251.726	188.972	154.618	78.204	205.3759
78	324.248	268.075	159.818	119.334	85.63	191.4209
80	335.193	264.608	195.661	164.555	124.257	216.8549
90	348.55	263.38	207.721	179.306	112.074	222.2062

Table 4.1: Threshold test average

threshold	4x4	5x5	6x6	7x7	8x8	died
60	9.05	14.5	19.5	24.9	27.8	19.15
70	1.6	4.8	6.7	8.7	9.7	6.30
72	2.25	3.6	6	9.1	11.6	6.51
74	1.8	3.35	6.4	6.5	10.3	5.67
76	1.8	4.2	5	5.8	9.7	5.30
78	2.25	3.55	6	8.9	8.4	5.82
80	0	0	0	0	0	0.00
90	0	0	0	0	0	0.00

Table 4.2: Percentage of dead agent

Chapter 5

CONCLUSIONS

The main goal of the assignment was to increase the performance of the agent regarding its reward and also regarding the time/space efficiency of the search.

The points that influenced the performance the most to get a high reward were an effective strategy and a good value for the risk threshold. We started with a general strategy and focused on corner cases and improved our strategy step by step, always trying to risk as less as possible, since dying has a cost of 1000. We tried our agent on many different random worlds, and our average result was always >0, which means that the agent is more successfull in finding the gold then dying or climbing out of the world. Most of the time the result was between 90 - 300, with an average of 200. In fact our simple analysis shows that lower risk thresholds are only better in small worlds.

In order to keep track of all the changes of the knowledge-base, the decisions made by the agent and the state of the agent, we developed the same class of the agent with loggers. These loggers write all the information on different log-levels to the log-file. Analyzing the log files, gave us the possibility to follow the execution of the search and thus test its correctness, but was also very helpful to compare different strategy approaches and choose the most effective one.

Regarding the execution time we tried 3 different approaches. The first one was only using the Bayesian network to infer pits locations and calculate their probability. Since we realized that the calculation of the probability, was very slow compared to the rest of the execution, we decided to reduce the workload of the computation of the Bayesian network, by introducing a SAT Solver to calculate the pit locations. From our simple empirical tests this showed a reduction in execution time of 5-10 %. Since the execution time was very short (around 0.1 seconds) and was also influenced by the background activity of our notebooks, we didn't had the chance to test this in a very accurate way. Such that a proper testing environment would be needed to determine, if the difference is significant or not. Furthermore, the efficiency of the SAT-solver depends on how often the Bayesian network is called. In simple worlds the SAT-Solver will be an overhead, cause it is never needed. The bigger the worlds and less the risk threshold is, the more Bayesian networks are called and thus performance with SAT increases. The last approach was to neither use Bayesian networks or SAT-solver and search only in the safe cells. This increased execution time by far and also the results where not bad. So if there is less computation power and memory available this would be for sure a good option.

Overall, we can conclude that we were able to find a very good strategy and performance of the agent. Nonetheless, a better analysis of the threshold and different implementation approaches, like SAT Solver, Planner could be tested and compared in order to further increase the performance.

Chapter 6

APPENDIX

threshold	stat_data	4x4	5x5	6x6	7x7	8x8
60	avg outcome	344.202	234.392	102.904	-14.722	-56.109
	std outcome	637.471	688.408	705.558	716.077	729.775
	grabbed gold	903	800	646	265	256
	percentage grabbed gold	45.15	40	32.3	26.5	25.6
	avg with gold	976.449	969.158	960.85	949.849	944.895
	std with gold	13.4131	18.3745	26.1252	34.3123	38.1691
	home without gold	916	910	964	486	466
	percentage home without gold	45.8	45.5	48.2	48.6	46.6
	avg without gold	-9.58188	-9.81429	-11.0788	-12.0247	-13.9785
	std without gold	45.8	45.5	48.2	48.6	46.6
	died	181	290	390	249	278
	percentage died	9.05	14.5	19.5	24.9	27.8
	avg died	-1019.63	-1026.24	-1036.46	-1046.54	-1048.52
	std died	13.2294	18.5087	26.4421	36.278	40.7984
70	avg outcome	334.069	256.19	182.768	120.336	65.394
	std outcome	503.667	538.931	542.198	543.37	527.008
	grabbed gold	729	645	544	231	190
	percentage grabbed gold	36.45	32.25	27.2	23.1	19
	avg with gold	976.981	970.27	963.752	958.719	946.737
	std with gold	13.224	18.4962	23.8224	31.2315	37.0873
	home without gold	1239	1259	1322	682	713
	percentage home without gold	61.95	62.95	66.1	68.2	71.3
	avg without gold	-8.84504	-10.672	-13.4788	-13.9047	-15.3128
	std without gold	61.95	62.95	66.1	68.2	71.3
	died	32	96	134	87	97
	percentage died	1.6	4.8	6.7	8.7	9.7
	avg died	-1035.06	-1041.75	-1051.68	-1053.39	-1067.71
	std died	11.4644	15.8843	19.853	26.0906	34.6172
72	avg outcome	351.238	276.18	193.095	126.244	47.048
	std outcome	521.617	523.409	532.133	555.915	549.565
	grabbed gold	777	660	548	243	190
	percentage grabbed gold	38.85	33	27.4	24.3	19
	avg with gold	977.453	970.352	963.175	954.749	949.5
	std with gold	12.77	19.1874	24.5127	32.2817	37.3984
	home without gold	1178	1268	1332	666	694
	percentage home without gold	58.9	63.4	66.6	66.6	69.4
	avg without gold	-8.89219	-10.2445	-12.0683	-13.6982	-13.9409
	std without gold	58.9	63.4	66.6	66.6	69.4
	died	45	72	120	91	116
	percentage died	2.25	3.6	6	9.1	11.6
	avg died	-1034	-1042.81	-1046.29	-1061.95	-1066.22
	std died	11.0272	15.426	19.2952	29.4224	36.7006

Table 6.1: Treshold test average

threshold	stat_data	4x4	5x5	6x6	7x7	8x8
74	avg outcome	337.555	265.522	180.433	153.147	60.656
	std outcome	508.672	514.253	534.474	522.36	535.23
	grabbed gold	740	635	266	243	192
	percentage grabbed gold	37	31.75	26.6	24.3	19.2
	avg with gold	977.08	968.791	963.259	956.025	946.786
	std with gold	13.6277	18.8865	23.9113	32.4783	39.4009
	home without gold	1224	1298	670	692	705
	percentage home without gold	61.2	64.9	67	69.2	70.5
	avg without gold	-8.75899	-10.8875	-12.6866	-14.6488	-16.2085
	std without gold	61.2	64.9	67	69.2	70.5
	died	36	67	64	65	103
	percentage died	1.8	3.35	6.4	6.5	10.3
	avg died	-1033.58	-1044.87	-1051.47	-1062	-1065.05
	std died	11.3514	16.9617	23.0407	26.8357	34.5553
76	avg outcome	353.359	251.726	188.972	154.618	78.204
	std outcome	512.805	525.825	511.636	507.348	537.795
	grabbed gold	771	623	259	236	203
	percentage grabbed gold	38.55	31.15	25.9	23.6	20.3
	avg with gold	977.565	971.311	966.008	956.208	949.926
	std with gold	13.1233	18.8543	23.8457	31.4918	38.1301
	home without gold	1193	1293	691	706	700
	percentage home without gold	59.65	64.65	69.1	70.6	70
	avg without gold	-8.19363	-10.9559	-12.5499	-13.8725	-15.4971
	std without gold	59.65	64.65	69.1	70.6	70
	died	36	84	50	58	97
	percentage died	1.8	4.2	5	5.8	9.7
	avg died	-1033.61	-1041.76	-1051.04	-1056.09	-1069.93
	std died	11.5813	15.6149	20.978	27.6582	36.5697
78	avg outcome	324.248	268.075	159.818	119.334	85.63
	std outcome	514.379	518.983	513.435	546.989	517.661
	grabbed gold	722	643	240	233	198
	percentage grabbed gold	36.1	32.15	24	23.3	19.8
	avg with gold	978.201	969.779	963.587	956.129	946.652
	std with gold	13.0488	18.0589	23.7067	30.9695	39.3677
	home without gold	1233	1286	700	678	718
	percentage home without gold	61.65	64.3	70	67.8	71.8
	avg without gold	-9.10868	-10.486	-12.3229	-13.705	-16.3343
	std without gold	61.65	64.3	70	67.8	71.8
	died	45	71	60	89	84
	percentage died	2.25	3.55	6	8.9	8.4
	avg died	-1034.11	-1041.31	-1046.95	-1057.89	-1072.37
	std died	11.3475	14.1507	18.8719	26.797	39.3972

Table 6.2: Treshold test average (Part 2)

threshold	stat_data	4x4	5x5	6x6	7x7	8x8
80	avg outcome	335.193	264.608	195.661	164.555	124.257
	std outcome	472.291	444.088	404.449	383.015	348.165
	grabbed gold	700	561	214	184	145
	percentage grabbed gold	35	28.05	21.4	18.4	14.5
	avg with gold	978.553	975.412	969.869	969.685	967.483
	std with gold	12.1502	14.5894	18.3013	21.1937	23.359
	home without gold	1300	1439	786	816	855
	percentage home without gold	65	71.95	78.6	81.6	85.5
	avg without gold	-11.2308	-12.5017	-15.1285	-16.9939	-18.7462
	std without gold	65	71.95	78.6	81.6	85.5
	died	0	0	0	0	0
	percentage died	0	0	0	0	0
	avg died	NaN	NaN	NaN	NaN	NaN
	std died	NaN	NaN	NaN	NaN	NaN
90	avg outcome	348.55	263.38	207.721	179.306	112.074
	std outcome	475.793	443.721	413.461	394.066	333.808
	grabbed gold	725	558	227	200	133
	percentage grabbed gold	36.25	27.9	22.7	20	13.3
	avg with gold	979.28	976.24	969.758	966.03	961.737
	std with gold	11.6444	14.2969	18.4118	24.5953	27.7827
	home without gold	1275	1442	773	800	867
	percentage home without gold	63.75	72.1	77.3	80	86.7
	avg without gold	-10.1012	-12.4695	-16.0595	-17.375	-18.2664
	std without gold	63.75	72.1	77.3	80	86.7
	died	0	0	0	0	0
	percentage died	0	0	0	0	0
	avg died	NaN	NaN	NaN	NaN	NaN
	std died	NaN	NaN	NaN	NaN	NaN

Table 6.3: Treshold test average (Part 3)