Assignment 01: 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: 12/11/2020

Academic year: 2020-2021

ABSTRACT

The task of the assignment was to implement an informed player to solve the "Hunt the Wumpus" game in the most efficient way using offline search algorithms.

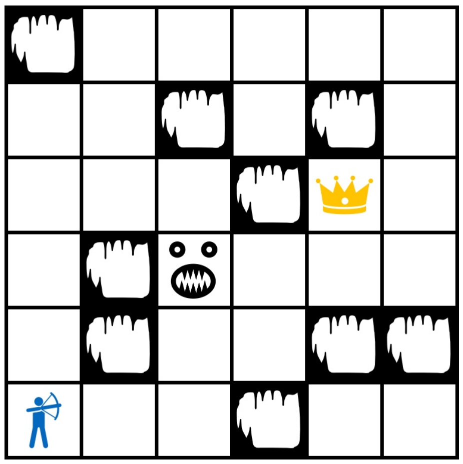
A brief description of the game is the following:
we are in a chess-like world environment where each box can either be empty or a pit or a block. The agent can only move in empty boxes and the movement is restricted by the orientation the agent has before performing the movement action. The world contains a gold, which has to be grabbed by the agent if that is possible, and a monster (wumpus) which can be killed if necessary in order to move in its 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 grab the gold and 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.
One thing to consider is that **there may not be a sequence** of actions to reach the gold, in that case the agent should climb out immediately from the world.

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.

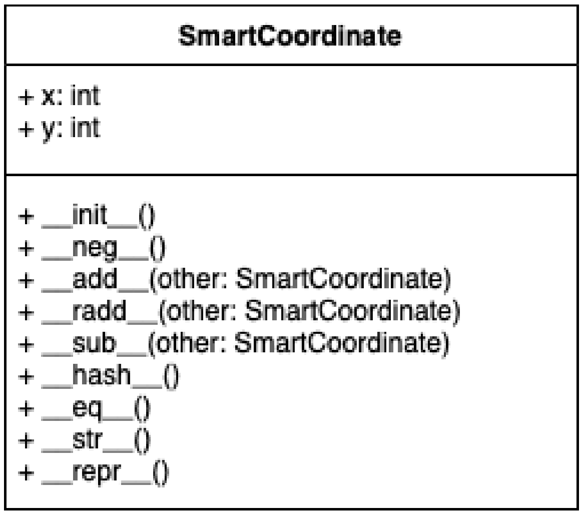
We implemented the operator (*SmartCoordinate* + *SmartVector*) so that you can sum a location with an orientation and get a *SmartCoordinate* representing the location of the tile you're moving into following that orientation.
We also defined some other helper methods. A noteworthy one is the ***get_perpendicular_vector_clockwise()*** 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 HuntWumpusState

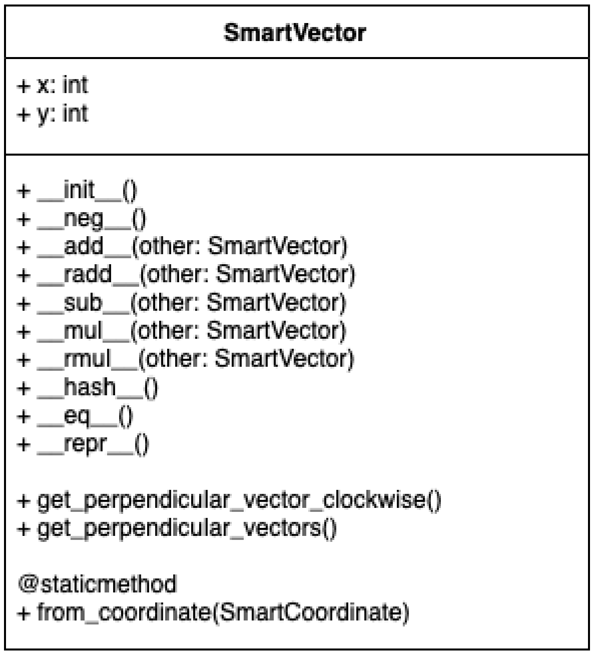
Focusing on the problem, we modeled the state of the problem in the *HuntWumpusState* class, defining the mutable data as attributes and the immutable data as static properties, as they won't change during the game so there's no need to keep them in memory for each state object.
The definition of the class is shown in Figure 2.2.

2.3 HuntWumpusNode

Then, we modeled the definition of a node for the problem in the *HuntWumpusNode* class to perform a graph-state search.
Each node represents a possible **state** for the game and has a reference to the **previous action** that lead to the current node and a reference to the parent node so that it can unwrap this chain and get the full sequence of actions from the initial state to the node itself.
Each node also has a **path_cost** which is the total actions' cost to get to that state and a **reward** which is the "points" the agent has accumulated up until the current state.
The definition of the class is shown in Figure 2.3.


SmartCoordinate
+ x: int
+ y: int
+ __init__()
+ __neg__()
+ __add__(other: SmartCoordinate)
+ __radd__(other: SmartCoordinate)
+ __sub__(other: SmartCoordinate)
+ __hash__()
+ __eq__()
+ __str__()
+ __repr__()

(a) *SmartCoordinate class diagram*


SmartVector
+ x: int
+ y: int
+ __init__()
+ __neg__()
+ __add__(other: SmartVector)
+ __radd__(other: SmartVector)
+ __sub__(other: SmartVector)
+ __mul__(other: SmartVector)
+ __rmul__(other: SmartVector)
+ __hash__()
+ __eq__()
+ __repr__()
+ get_perpendicular_vector_clockwise()
+ get_perpendicular_vectors()
@staticmethod
+ from_coordinate(SmartCoordinate)

(b) *SmartVector class diagram*

Figure 2.1: Linear Space classes

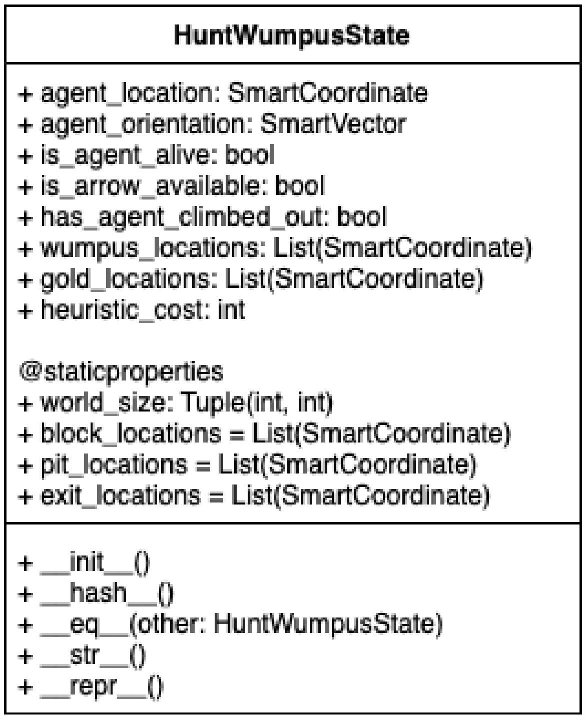

HuntWumpusState
+ agent_location: SmartCoordinate
+ agent_orientation: SmartVector
+ is_agent_alive: bool
+ is_arrow_available: bool
+ has_agent_climbed_out: bool
+ wumpus_locations: List(SmartCoordinate)
+ gold_locations: List(SmartCoordinate)
+ heuristic_cost: int
@staticproperties
+ world_size: Tuple(int, int)
+ block_locations = List(SmartCoordinate)
+ pit_locations = List(SmartCoordinate)
+ exit_locations = List(SmartCoordinate)
+ __init__()
+ __hash__()
+ __eq__(other: HuntWumpusState)
+ __str__()
+ __repr__()

Figure 2.2: HuntWumpusState class diagram

2.4 HuntWumpusProblem

Eventually, we modeled the formal definition of the problem in the *HuntWumpusProblem* class, defining:

- the **transition model** of the world
- the **initial state**
- the **goal test**
- the **path cost** func

We went a step further by introducing the *get_effective_actions(for_state)* and *get_best_actions(for_state)* methods, which **filter out the useless actions** for the current state. The *get_effective_actions(for_state)* performs the first-level filtering: it filters out all actions that applied to the current state will result in the exact same state.
The *get_best_actions(for_state)* performs a second level filtering: it takes all effective actions and filters out the useless ones, such as shooting if in the neighbour location there is no wumpus, does the grabbing in a location where there is no gold, or climbing out if we haven't grabbed the gold yet. It also analyses the neighbouring locations checking if they're blocks, pits, or the

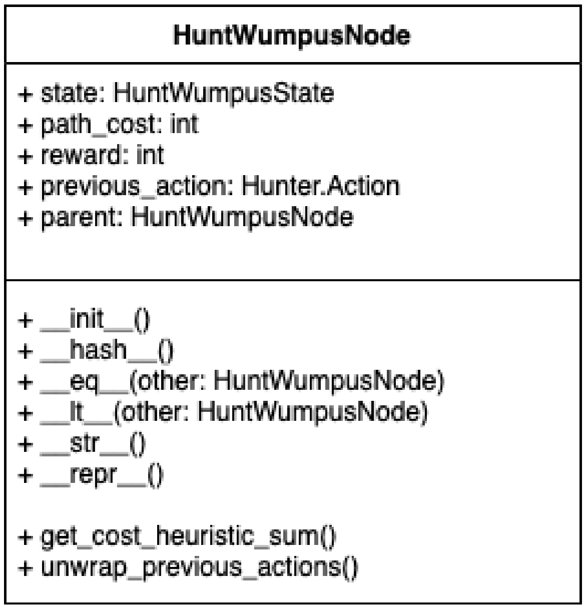

HuntWumpusNode
+ state: HuntWumpusState
+ path_cost: int
+ reward: int
+ previous_action: Hunter.Action
+ parent: HuntWumpusNode
+ __init__()
+ __hash__()
+ __eq__(other: HuntWumpusNode)
+ __lt__(other: HuntWumpusNode)
+ __str__()
+ __repr__()
+ get_cost_heuristic_sum()
+ unwrap_previous_actions()

Figure 2.3: HuntWumpusNode class diagram

border of the map, and only returns the cheapest rotation actions to reach all legal neighbour locations since there's no advantage in rotating to a location facing an obstacle or performing three RIGHT actions instead of a single LEFT action. One last thing it does is that it returns an empty list of actions if the gold is in a pit location, since we know there are no sequence actions to reach the goal.
The actual definition of the class is shown in Figure 2.4.

We also introduced the named tuple *HuntWumpusResult* to carry around the result of the search algorithm. It just has two attributes: *sequence_actions* (which is the list of actions to reach the goal starting from the initial state) and *total_reward* (which is the difference between the reward of the last node and the *path_cost* to reach it).

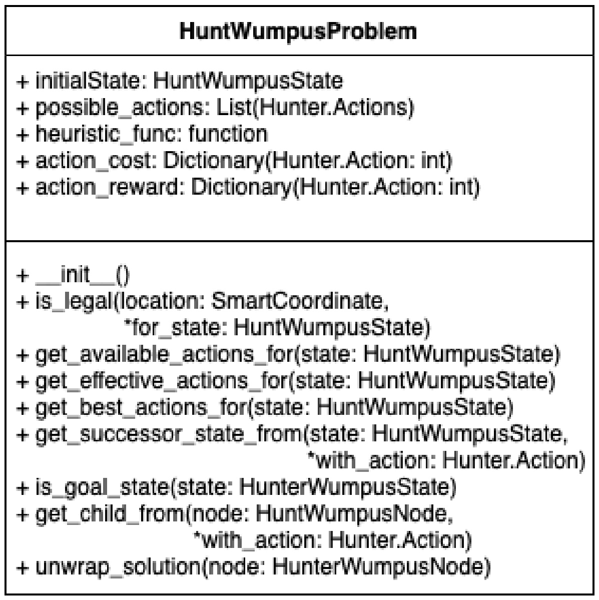

HuntWumpusProblem
+ initialState: HuntWumpusState
+ possible_actions: List(Hunter.Actions)
+ heuristic_func: function
+ action_cost: Dictionary(Hunter.Action: int)
+ action_reward: Dictionary(Hunter.Action: int)
+ __init__()
+ is_legal(location: SmartCoordinate,
*for_state: HuntWumpusState)
+ get_available_actions_for(state: HuntWumpusState)
+ get_effective_actions_for(state: HuntWumpusState)
+ get_best_actions_for(state: HuntWumpusState)
+ get_successor_state_from(state: HuntWumpusState,
*with_action: Hunter.Action)
+ is_goal_state(state: HunterWumpusState)
+ get_child_from(node: HuntWumpusNode,
*with_action: Hunter.Action)
+ unwrap_solution(node: HunterWumpusNode)

Figure 2.4: HuntWumpusProblem class diagram

Chapter 3

SEARCHING

We started by implementing the **Uniform Cost Search** (UCS) searching algorithm, which is available in the `hunt_wumpus_UCS.py` file. We followed the implementation of the pseudocode for the UCS AIMA4e found on: `https://github.com/aimacode/aima-pseudocode/blob/master/md/Uniform-Cost-Search.md`.

We made some tests to check the correctness of the implemented solution and then moved to the implementation of the **A*** searching algorithm, which is just a variant of the UCS taking into consideration the value of a heuristic function. The code of the A* searching algorithm can be found on `hunt_wumpus_AStar.py` file.

3.1 Breaking ties

Since we are working in a grid map environment there a lot of paths with the same length reaching the same destination. If we don't introduce a tie break, we'll end up exploring all nodes with the same value before moving forward to a more promising node. We implemented the *lower-than* ($<$) operator for the *HuntWumpusNode* to define an order of exploration for nodes in the priority queue with the same (*path_cost* + *heuristic_cost*) value. We define the following hierarchy for the order:

1. the node with the lowest heuristic is explored first

2. the node with the state that is closer to the goal is explored first (the one with the least Manhattan distance between the *agent_location* and the *goal_location* for that node)

If all previous conditions are not met, we defined the following hierarchy for the orientation, $N > E > W > S$ (so the node pointing North will be explored first and so on).

Chapter 4

HEURISTIC FUNCTIONS

Let's talk about the fun stuff, we started developing easy heuristic functions to catch all the smart ideas we got.
For instance, if there is a gold in the world then I need to grab it for sure in the future (+ 1 to the cost) whereas if there's no gold it means I already grabbed (+ 0 to the cost). The same goes for the climbing, if I haven't yet climbed out I need to do it once at least (+ 1) otherwise I don't need it (+ 0).

Since we need to go straight to the gold and then back to the exit location, we developed heuristics functions which only takes care of the "goal" location to reach at that moment. If we haven't grabbed the gold yet, we consider the goal location to be the gold location and then we add to the *base_cost* the *manhattan_distance* from the gold location to the exit location. Otherwise, if we've already grabbed the gold, then we consider the goal location to be the exit location without adding anything to the *base_cost*.
One last consideration we made was that if the wumpus was in the same location as the gold or exit locations, then we would need to kill it for sure, so we eventually added this cost to the *base_cost*.

We developed the heuristic functions taking all preceding observations into account - from now on, the *base_cost(*)* variable will represent all these observations - and they are all available in the `heuristic_functions.py` file.
A brief description of each of them is available below.

4.1 heuristic_func_manhattan

This function calculates the Manhattan distance from the agent location to the goal location and returns the distance to reach the goal. We developed this heuristic for a testing and comparison purpose, so we didn't include the observations introduced before.
An example of the calculation of this heuristic function can be found in Figure 4.1.

4.2 heuristic_func_manhattan_with_orientation_overhead

Is a very simple approach we attempted, and it actually worked a bit. With this heuristic, we are just returning the Manhattan distance from the agent location to the goal plus the minimum number of rotations the agent has to do to reach the goal location. It first calculates the cost to make the agent pointing to the goal location. Then, if the goal is North, East, South, or West,

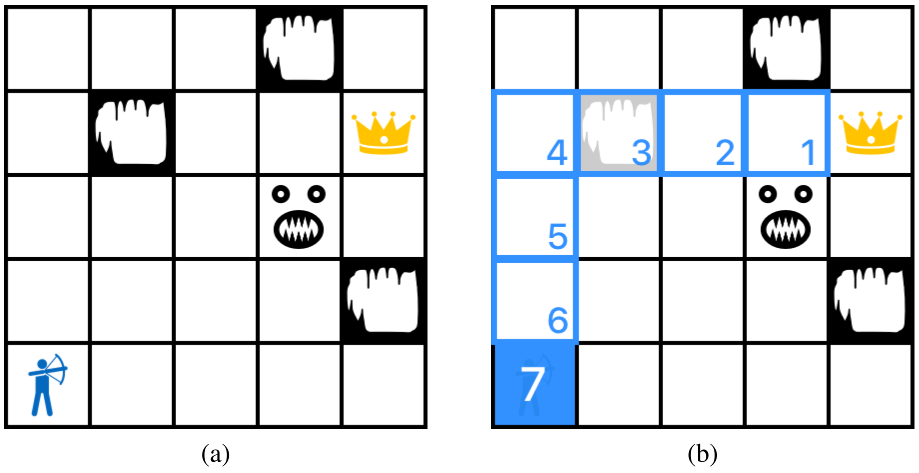


Figure 4.1: Manhattan distance heuristic calculation example

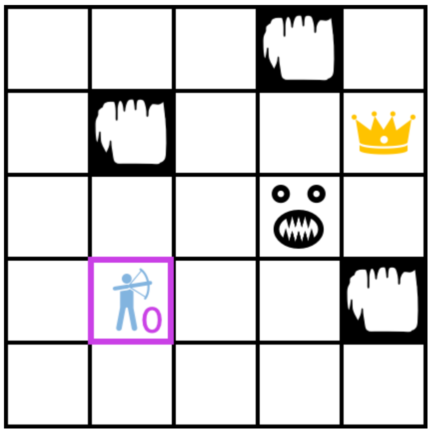
then we could go straight if there are no obstacles between the goal and we are pointing in that direction. Otherwise, if the goal is North-East, North-West, South-East or South-West then we know we need at least one rotation to reach it.
An example of the calculation of this heuristic function can be found in Figure 4.2.

4.3 heuristic_func_best_neighbour

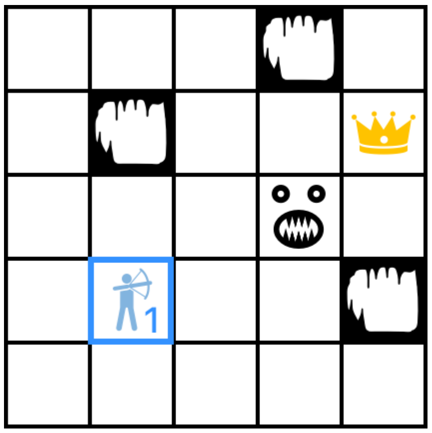
The basic idea for this heuristic is that it checks all neighbours and return the most promising one using the Manhattan distance from it. There's just a catch when the location of the agent is in the goal location: it will check the most promising neighbour without realising he reached the goal. So we introduced an early exit in the function which checks for this specific condition and return 0. The function will analyse all neighbour locations (the orthogonally adjacent ones) and will calculate the sum of:

- orientating to the neighbour
- shooting (if there's a wumpus in that location)
- moving to neighbour location
- Manhattan distance from neighbour to the goal
- orientation overhead to reach the goal from the neighbour[1]

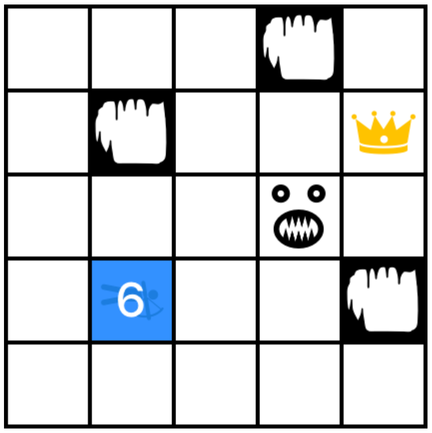
[1]the orientation overhead is calculate as follow: 0 if the goal direction is North, East, South or West (so it may need no rotations), 1 if the goal direction is North-East, North-Ovest, South-East, South-West (so at least he needs one rotation)



(a) *Cost to orientate in gold direction*



(b) *Cost of the rotation overhead to get the gold*



(c) *Sum of (a) and (b) with Manhattan distance*

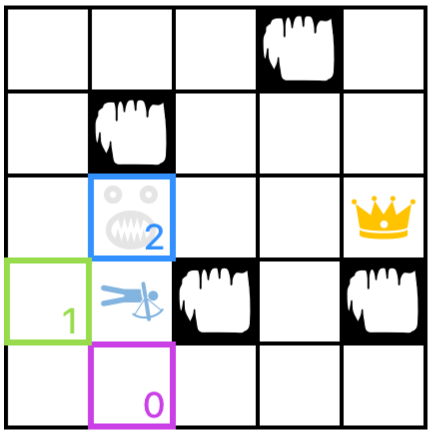
Figure 4.2: Manhattan distance with orientation overhead heuristic calculation example

In the end, we just return the minimum of them all.
An example of the calculation of this heuristic function can be found in Figure 4.3.

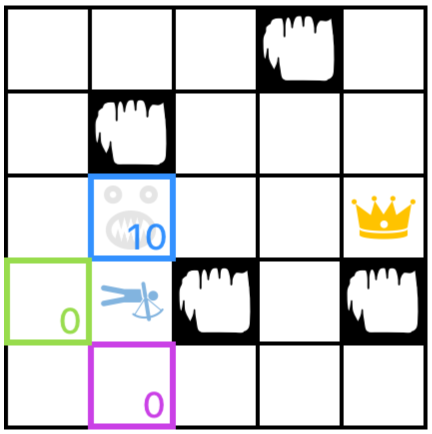
4.4 heuristic_func_smart_manhattan

We considered every possible attribute for the given environment, the gold, the wumpus, the climb out and so on. The only thing we could improve one step forward was the routing of the agent, and that's why we developed the *_smart_manhattan_distance* function.

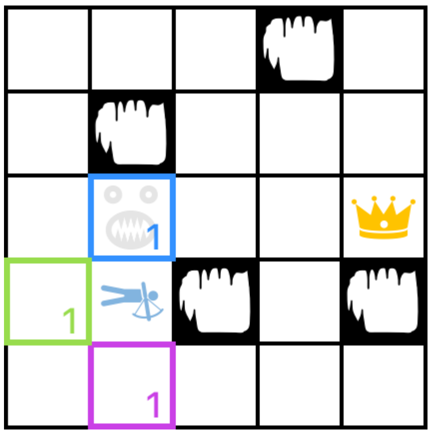
It takes a start location, a destination location and the list of blocks available in the grid world and returns the most precise number of steps + rotations needed for the agent to reach the destination.
The idea is that we look if there is a possible Manhattan distance path from the agent location to the destination location. If there is one, we scan for some patterns to calculate the rotations'



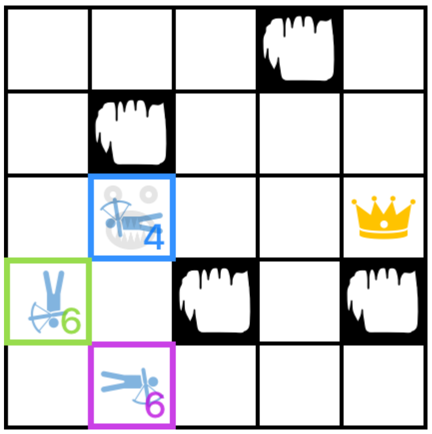
(a) *Cost to orientate to neighbour*



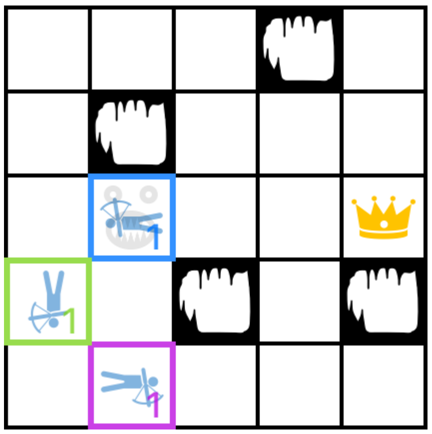
(b) *Cost of shooting if neighbour contains wumpus*



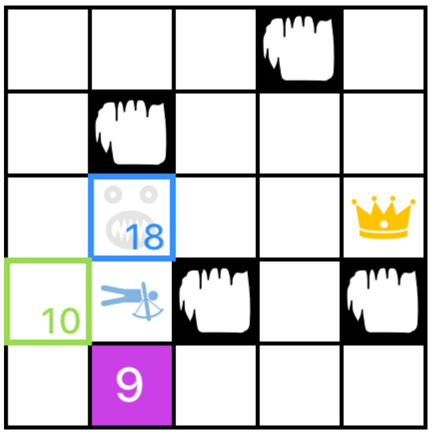
(c) *Cost of moving into the neighbour*



(d) *Value of Manhattan distance between neighbour and gold*



(e) *Cost to orientate to gold from neighbour*



(f) *Sum of values (a), (b), (c), (d), (e)*

Figure 4.3: Best neighbour heuristic calculation example

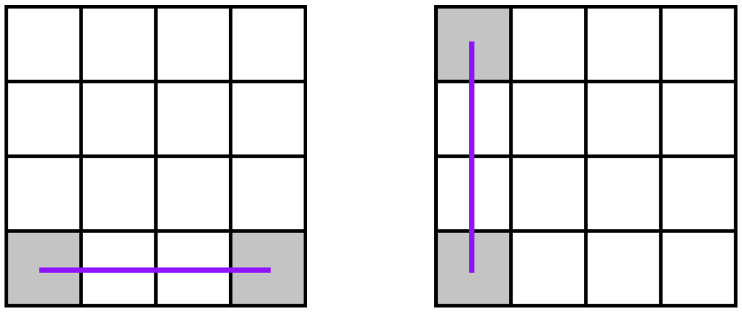

(a) *Moving straight horizontally* (b) *Moving straight vertically*

Figure 4.4: No rotation needed example cases

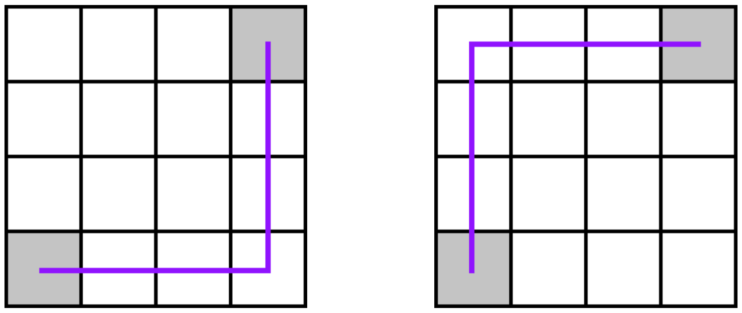
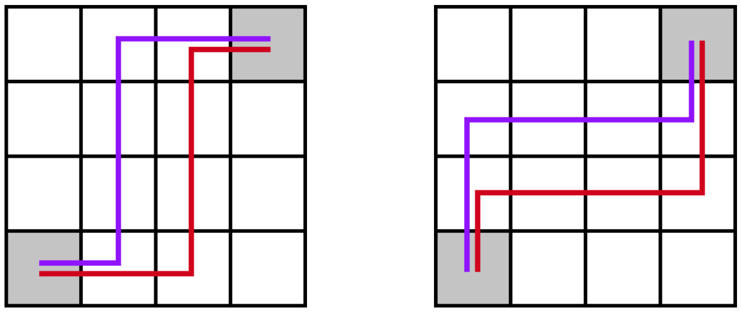

(a) *Moving straight East and then straight North* (b) *Moving straight North and then straight East*

Figure 4.5: One rotation needed example cases

overhead to follow that path.

- no rotation needed (Figure 4.4)
  - moving straight in just one direction
- one rotation needed (Figure 4.5)
  - moving straight East and then straight North
  - moving straight North and then straight East
- two rotation needed (Figure 4.6)
  - moving straight exactly in one direction (either North or East)
- three or more rotations needed
  - all other cases



(a) *Moving straight only vertically* (b) *Moving straight only horizontally*

Figure 4.6: Two rotation needed example cases

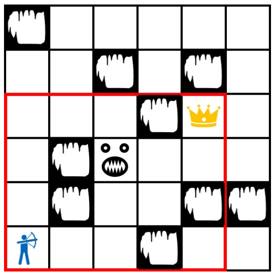
In this case, we return the Manhattan distance number of steps from start to destination + number of rotations needed to reach the destination. Otherwise, if there is not a Manhattan path to reach the goal, we return *manhattan_distance* number of steps from start to destination + 4, which is the cheapest case to get around, shown in the figure. In this case, since we're dealing with the heuristic function, we considered pits the same as blocks as we would die if entering one of such locations. We don't care about the orientation of the agent in the start location, we will take care of that as a further step in the heuristic function.

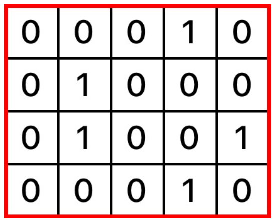
So, how does that *smart_manhattan_distace* function works?
It takes all relevant information regarding movement: the map size, the block locations, the start location and the destination location. Then it makes a binary mapping matrix considering blocks as 1s and empty locations as 0s. It then scans the matrix only moving Up and East at each step, starting from the bottom-left location. The workflow of the scanning is shown on Figure 4.7.

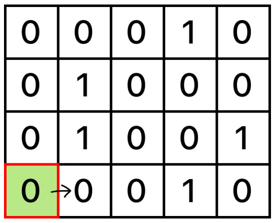
In the end, if the top-right location was found at the end of the scanning, then we know a possible path with just Manhattan distance steps is available, otherwise not. In case of success, we eventually scan for pattern matching (described below on Figure 4.7) and then return the Manhattan distance + rotations' overhead.

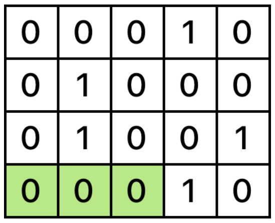
4.5 heuristic_func_best_neighbour_smart_manhattan

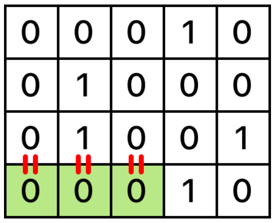
With the great results given by the previous heuristic, we decided to make a re-implementation of the *best_neighbour* heuristic function to combine the performance of the two altogether, and we came up with the best heuristic function we could develop.

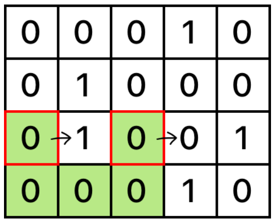

(a) *Taking the smallest area containing the agent and the gold*

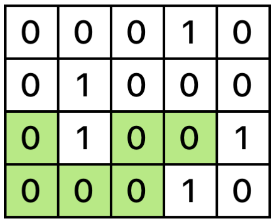

(b) *Converting the area in a binary matrix (1 for pit)*

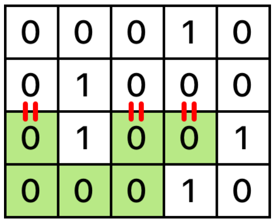

(c) *Moving right from initial location*

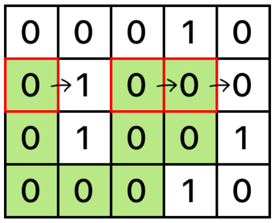

(d) *Reachable locations up to this iteration*

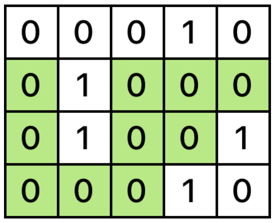

(e) *Moving up from previous row*

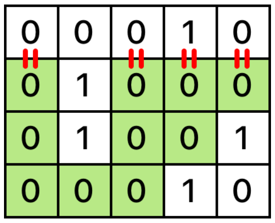

(f) *Moving right from reachable locations*

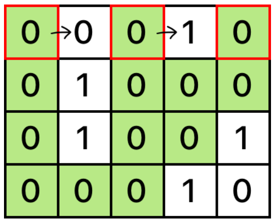

(g) *Reachable locations up to this iteration*

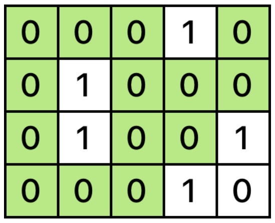

(h) *Moving up from previous row*


(i) *Moving right from reachable locations*


(j) *Reachable locations up to this iteration*


(k) *Moving up from previous row*


(l) *Moving right from reachable locations*


(m) *Reachable locations up to this iteration*

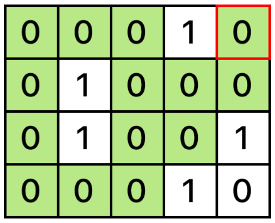

(n) *Resulting matrix*

Figure 4.7: Smart Manhattan distance workflow

Chapter 5

METHODS USED TO COMPARE ALGORITHMS

In order to evaluate the performance of the algorithms and heuristics we modelled different worlds, with different characteristics and position of wumpus, gold, agent and pit. Since the heuristics take in account different attributes of the world we expect that heuristics, will perform differently in each of them. Some heuristics like *heuristic_func_best_neighbour_smart_manhattan* are a combination of the other and therefore we expect this to be the best one.
The evaluation was performed by implementing a counter in the code that was increased by 1 each time a new node was extracted from the frontier and expanded.

In the following we describe the worlds with their particularities. Their relative graphic representation can be found in the A

- World 1: free world, there is only the agent and the gold;
- World 2: free world, there is only the agent, the gold and the wumpus;
- World 3: unsolvable world;
- World 4: there are pits, the agent has to pass through them;
- World 5: the wumpus and the gold are in the same location;
- World 6: the agent has to kill the wumpus for reaching the gold;
- World 7: it is more convenient for the agent to choose a path where there is no wumpus;
- World 8: it is more convenient for the agent to choose a path where there is the wumpus;

Chapter 6

CONCLUSIONS

As described in the previous chapters, we used different algorithms and heuristics to perform test-runs on examples worlds to measure their performance by counting the expanded nodes. As an **upper bound** for the informed search algorithms, we chose the uninformed **Uniform Cost Search Algorithm (UCS)** - pink bar in figure from 6.1 to 6.8. The **lower bound** is the **minimal number of actions needed** to reach the goal - black horizontal line in figure from 6.1 to 6.8.

For comparison purposes, we made also some test-runs with well-known uninformed search algorithms, like BFS and DFS. Since they don't guarantee to find the best solution, and mostly visited many more nodes than UCS or A*, we don't consider them in the next comparisons. It is clearly visible that already a very simple heuristics like the Manhattan Distance *heuristic_func_manhattan* improves the search algorithm by far.

A further enhancement of the search was taking care of the orientation, when either the gold or the exit location are not aligned to the hunter position. The difference to the *heuristic_func_manhattan* is little since the difference in the heuristic is max 3. The *heuristic_func_best_neighbour* is a further improvement of the previous heuristic function, because it checks all it's neighbours and returns the minimum heuristic cost between them and the cost to reach them. The *heuristic_func_smart_manhattan* takes another approach by recognizing patterns within the grid map and performs better than the previous ones. The last heuristic function *heuristic_func_best_neighbour_smart_manhattan* is a combination of all the previous and as expected outperforms the other heuristics and is **very close to the optimal** in these chosen worlds.

Summing up we can conclude that the **more information** about the world is added to the heuristic function **the better is their performance**, when comparing the amount of explored nodes. In general, there is a trade-off between the computation of the heuristics and the minor computation of the nodes. In our cases the better the heuristics the faster was the search algorithm in finding a solution.

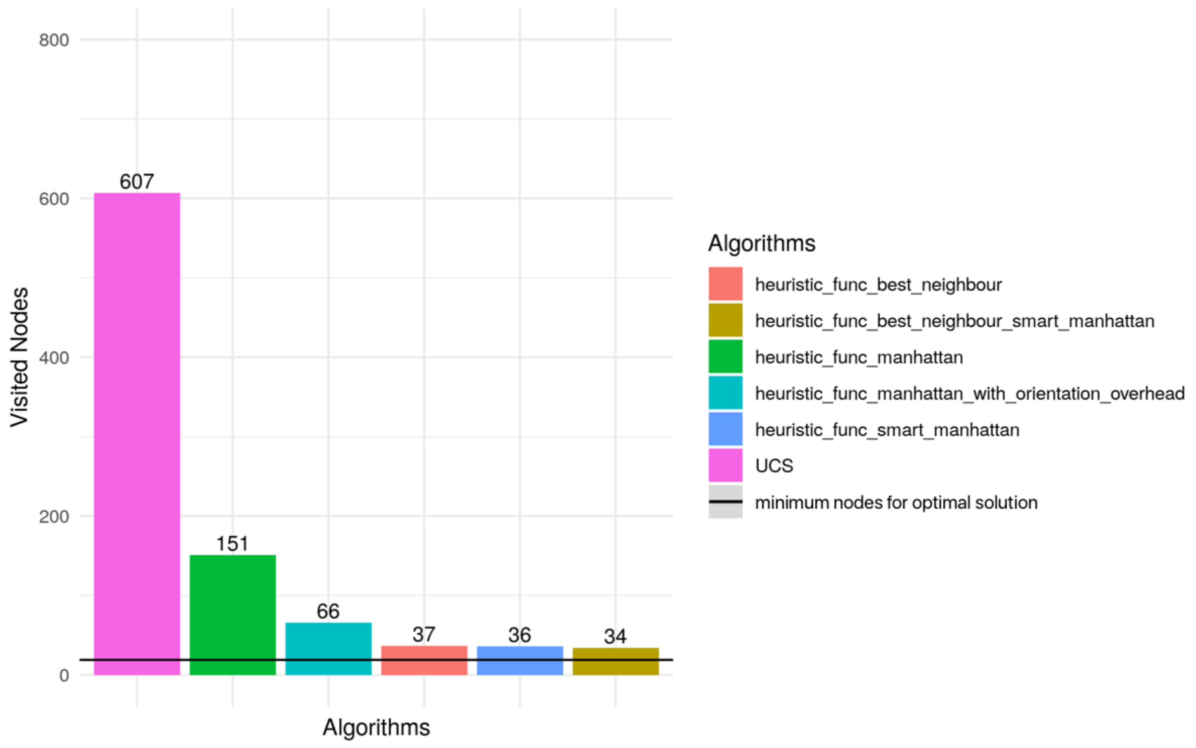


Figure 6.1: World 1: comparison of explored nodes

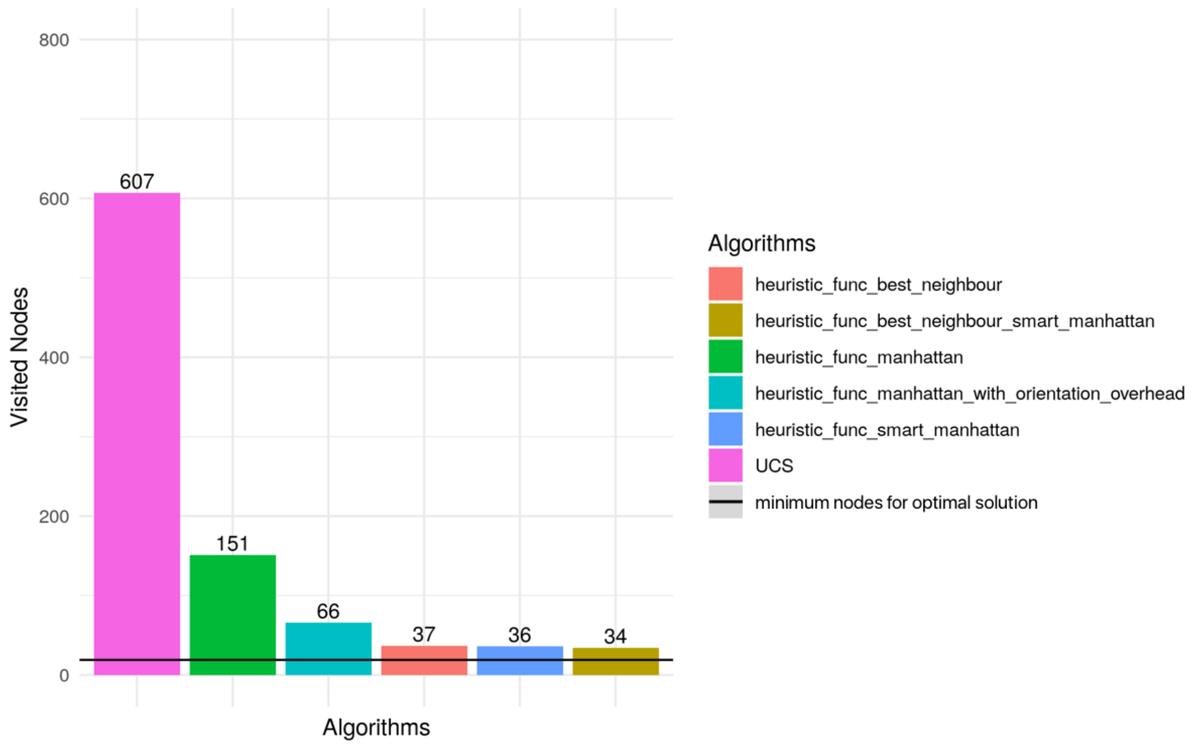


Figure 6.2: World 2: comparison of explored nodes

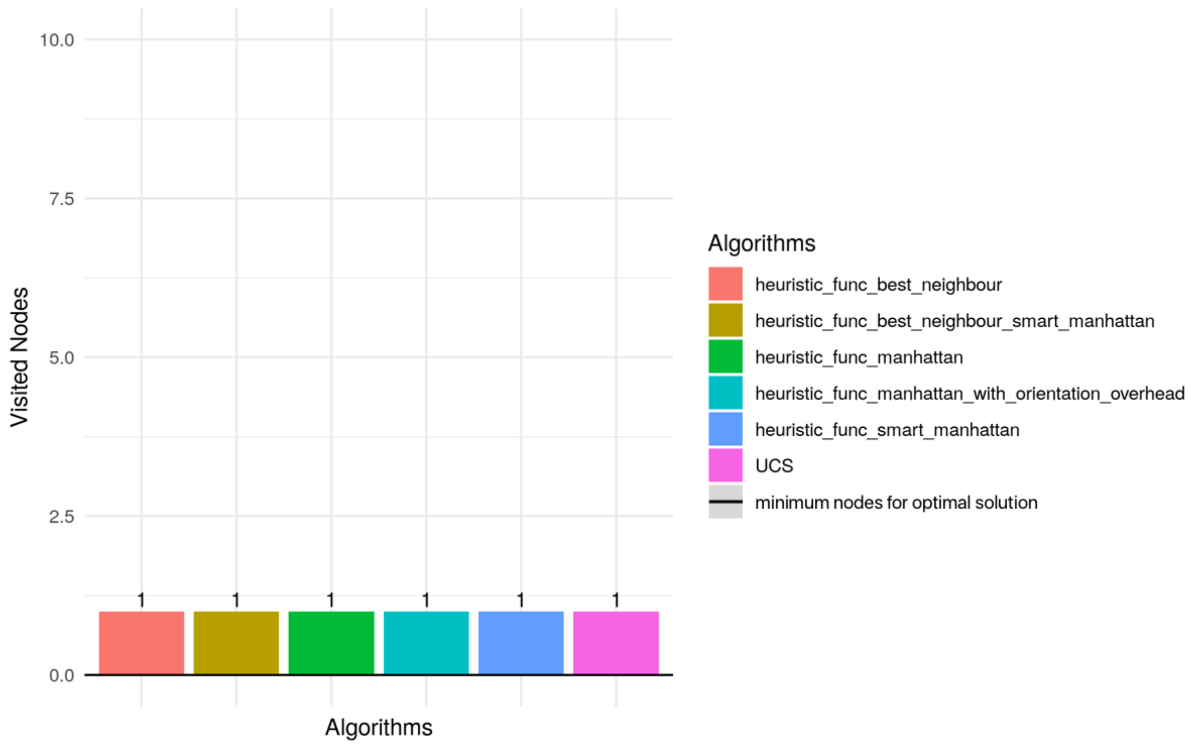


Figure 6.3: World 3: comparison of explored nodes

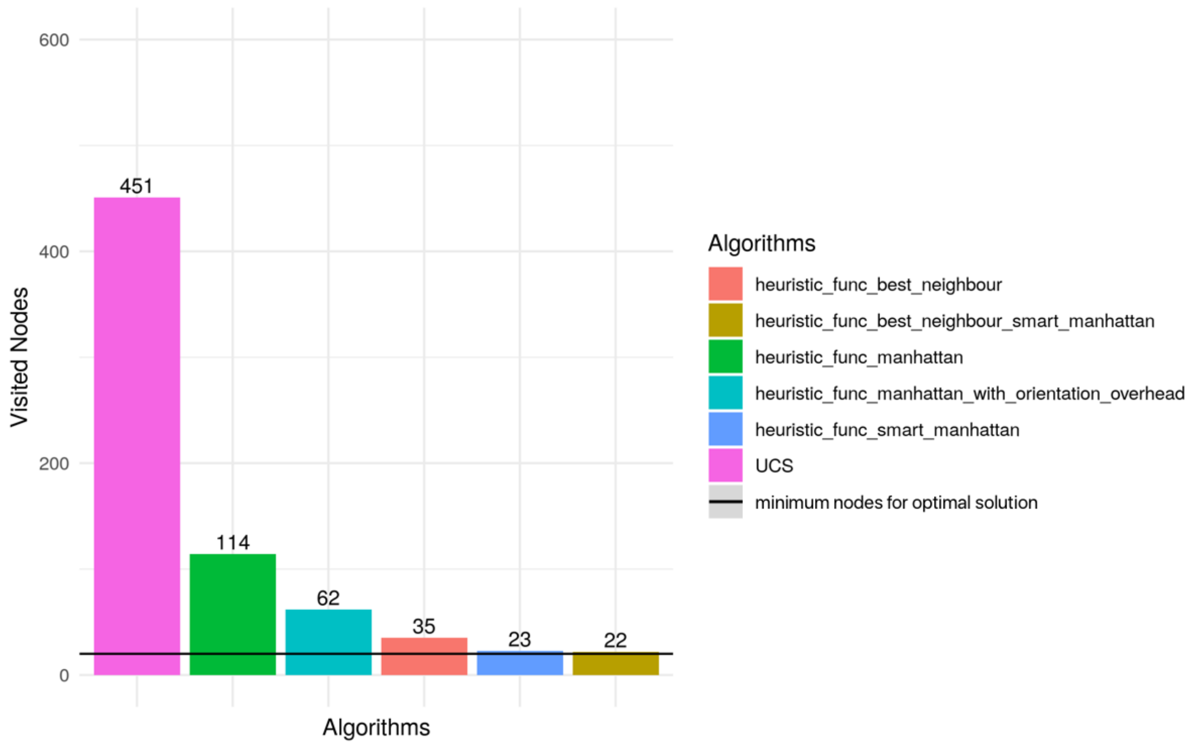


Figure 6.4: World 4: comparison of explored nodes

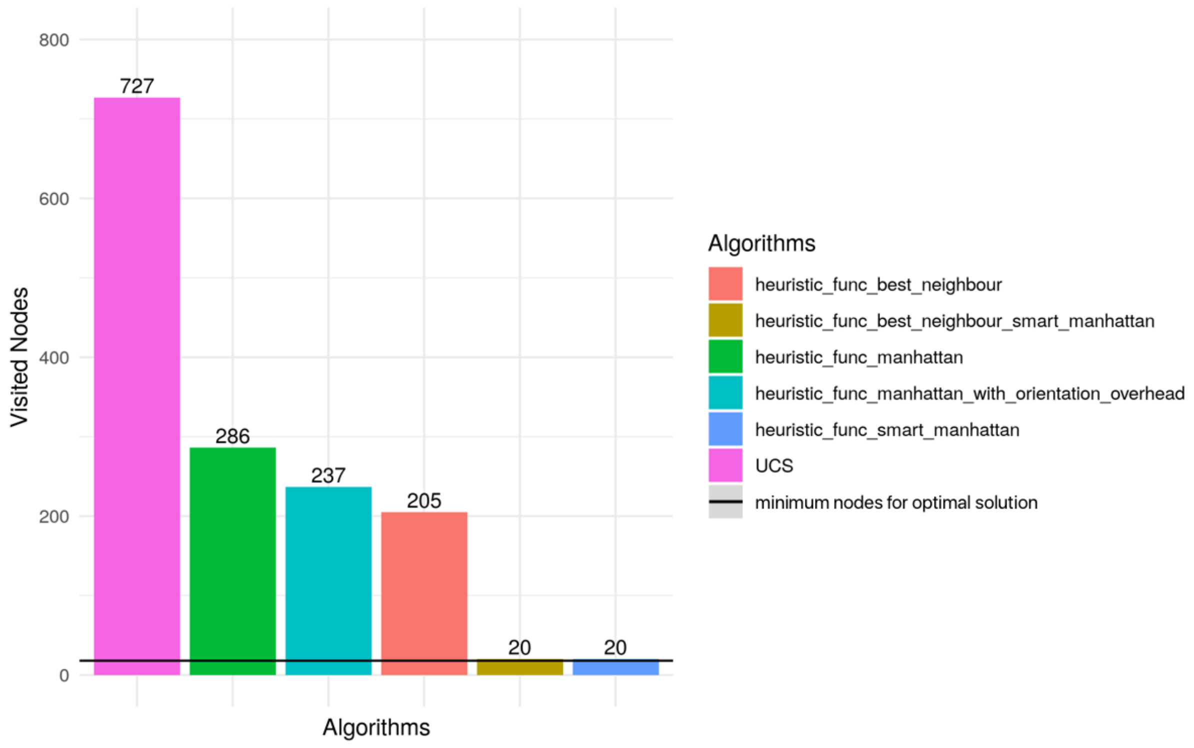


Figure 6.5: World 5: comparison of explored nodes

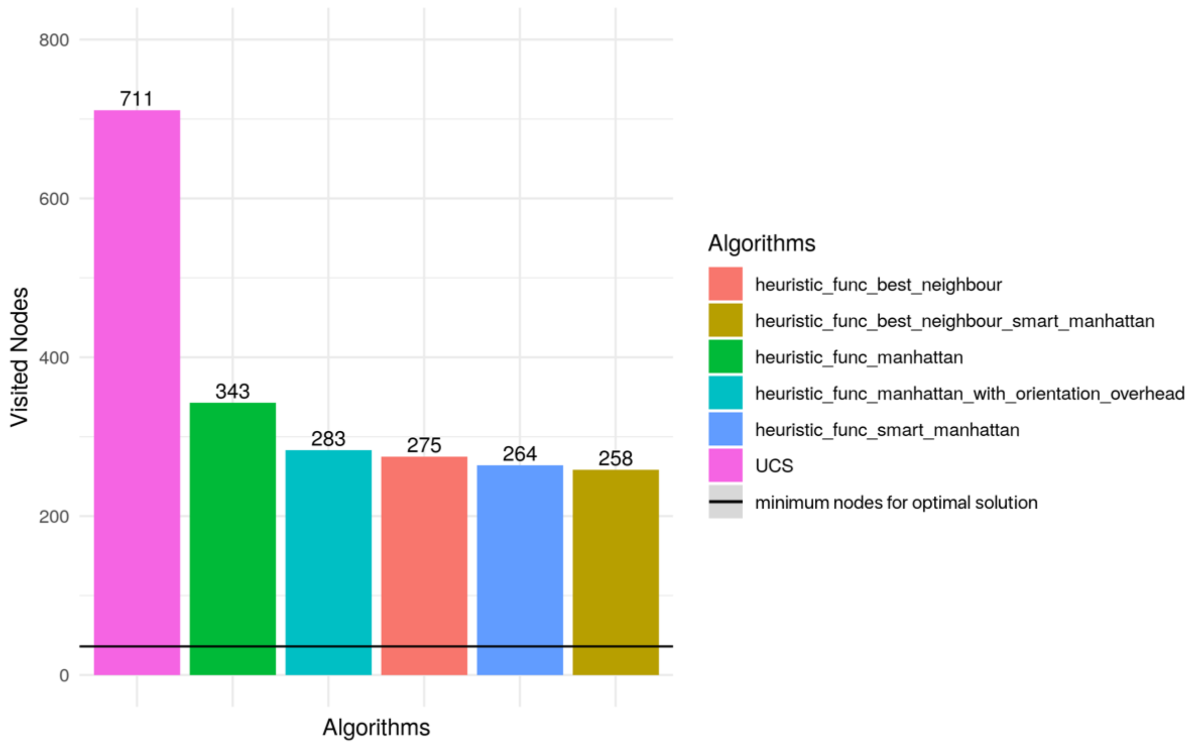


Figure 6.6: World 6: comparison of explored nodes

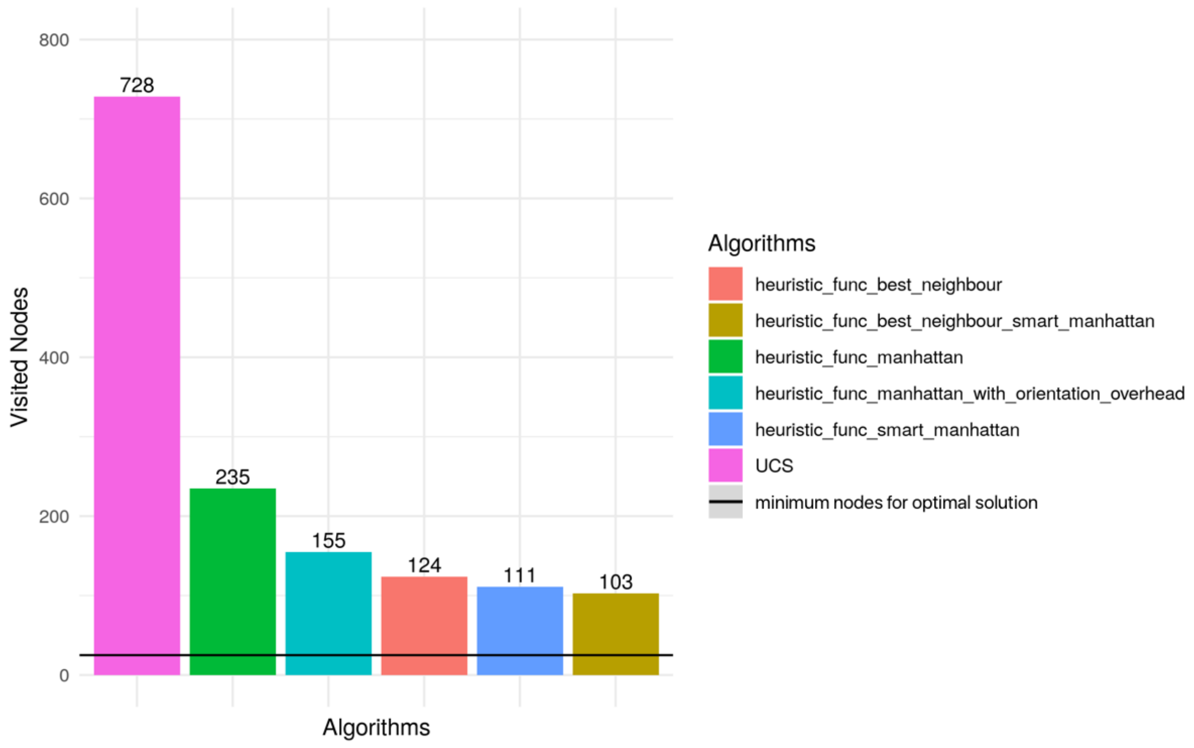


Figure 6.7: World 7: comparison of explored nodes

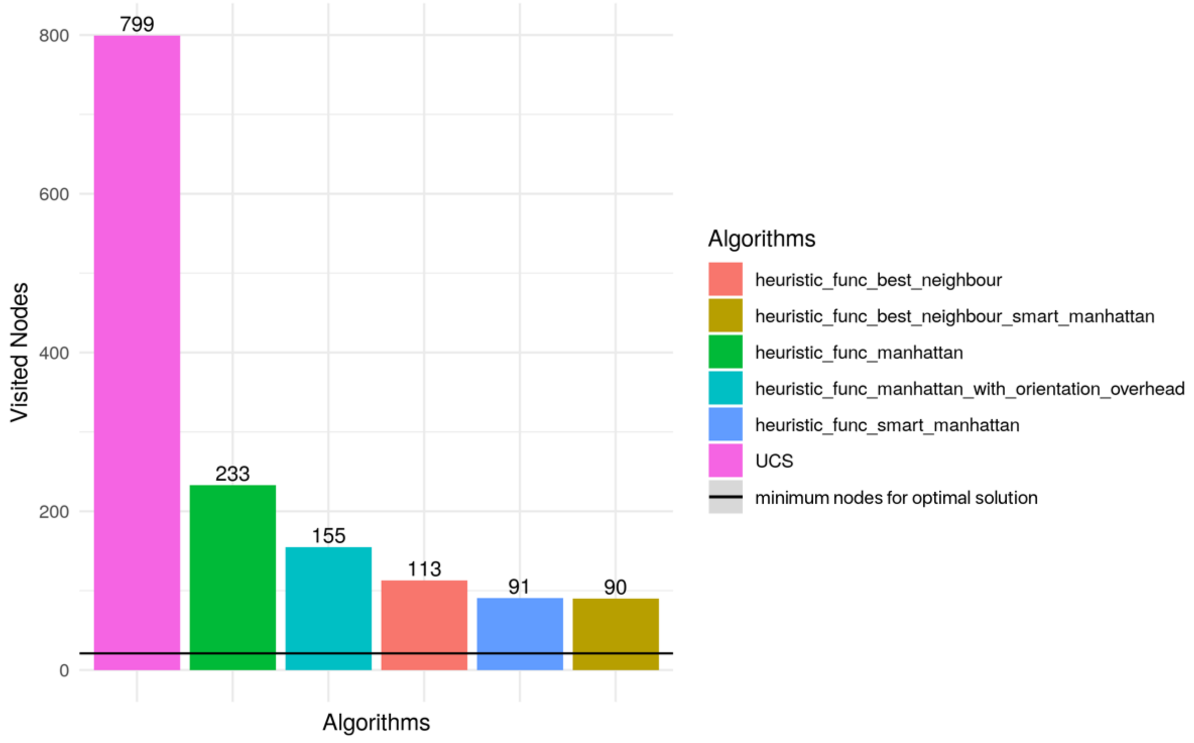
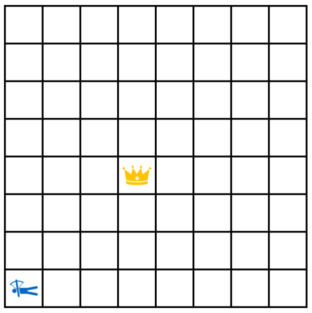


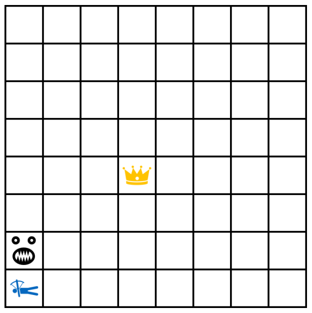
Figure 6.8: World 8: comparison of explored nodes

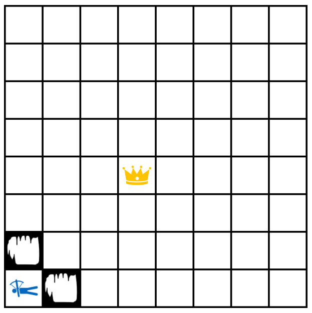
Appendix A

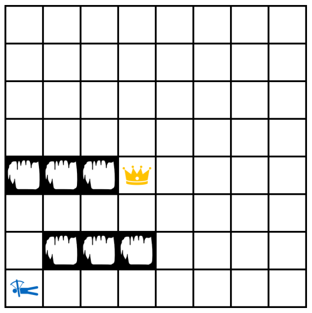
Graphic representation of the worlds in chapter 5.

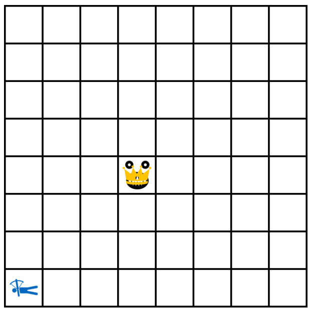
- World 1: free world, there is only the agent and the gold;
- World 2: free world, there is only the agent, the gold and the wumpus;
- World 3: unsolvable world;
- World 4: there are pits, the agent has to pass through them;
- World 5: the wumpus and the gold are in the same location;
- World 6: the agent has to kill the wumpus for reaching the gold;
- World 7: it is more convenient for the agent to choose a path where there is no wumpus;
- World 8: it is more convenient for the agent to choose a path where there is the wumpus;

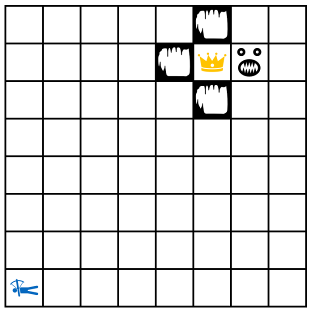

(a) *World 1*

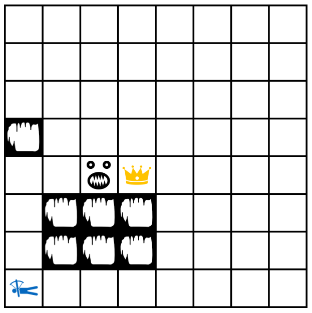

(b) *World 2*


(c) *World 3*


(d) *World 4*


(e) *World 5*


(f) *World 6*


(g) *World 7*

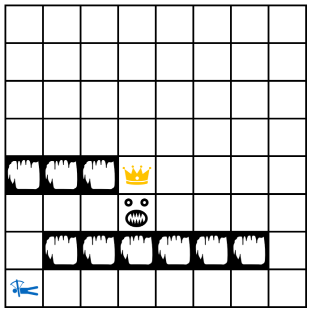

(h) *World 8*

Figure A.1: Test worlds