Kevin Langer
Assignment 2
CS4100

A2 Documentation

The basic structure of A2 is a system that generates rules for first order logic rules and submits them to prover9. The program queries relevant squares for a yes or no answer on the safety of a square or the proof that there is a wumpus in the square. The agent does all it can to maximize its score. With gold being worth 1000, step being -1, and the arrow being -100. The general strategy adapted is to be as greedy as possible. The expected value on the score of moving to a new score is completely unknown with a ceiling of +999 and a floor of -101. While the median for traveling around is likely -1, the true value is unknown. As a result, it is assumed that gold could appear anywhere in the map. The agent exhaustively searches all safe squares and kills any wumpus for which it can deduce its location. Only after it runs out of safe squares and wumpus to kill does it navigate to its goal.

Goal navigation is done using A* search. A* is a complete search algorithm that is guaranteed to find the shortest path through the map of known safe squares. Here the safe squares do **not** have to be visited; they only have to be proven safe by the first order logic engine. As a result, after every move the A* search is preformed again. This is because a list of new safe squares could be updated on a move. As a result the A* algorithm is a standard one, but only the first move from the path is ever used. While the A* is fairly straight forward. The job of picking the goal is not. Not only does the agent need to travel to a safe square as quickly as possible, but it also needs to pick *which safe square* to go to out of a list of safe and unvisited cells. This forms a simplified traveling salesman problem (different from the TSP because we can revisit nodes). For this problem, I do not find the optimal solution but instead use a heuristic to find the closest cell by the Manhattan distance. Because the safe unvisited squares are frequently close and most likely adjacent—where the breadth of unvisited safe squares is quite small—the heuristic preforms very similarly to the optimal solution but at a much greater use of computational resources.

First order logic proofs are performed using prover9 and automatically generated using the python framework. While the python code runs quickly, the prover9 code comes with significant performance overhead. It is important to note that many things done in the python framework were done to maximize performance. After being spoiled with a very powerful machine, and then later finish development on a 1Ghz laptop care was taken to reduce unnecessary computation. The most significant of these was to only calculate the safety of adjacent squares unless the wumpus has just been killed. Another performance increase came from reducing the amount of unnecessary facts added to the KB by separating a portion of the safety rules from the finding wumpus rules.

Generating the first order logic rules was particularly difficult. Prover9 provides no debugging of rules that are not proven. As a result, formation of these rules was difficult. Prover9 was only used for the inference part of the wumpus environment. Anything that did not require inference was done in python. The basic structure of proving safety of this was to define all adjacent squares as knowledge. State that all adjacent squares to a pit are implies breezy. Proving the possibility of a wumpus was done in the same way. Using this one way implication it is clear that the agent can never prove the existence of either a wumpus or pits with these rules; it can only prove the proximity of one. It turns out, due to the multiplicity of the pits it is impossible for the safe agent to prove that pits exist (for example, in a checkerboard the agent will believe there are pits in locations that are safe, due to pits adjacent to the safe square on all sides. Proving the wumpus is done later, with the assumption that there is only a maximum of one wumpus in the world. Safety is done by implying safe if there is no pit and no wumpus and safety on stenches and wumpus is also implied if a scream is heard from the arrow. Proving the existence of the wumpus requires additional rules. Specifically, that if there is a stench and a space is not adjacent to that stench, for all stenches, there is no wumpus in that space. Likewise, the fact that the wumpus can only be in adjacent spaces is given. Once a wumpus is detected and the agent is in place the game engine checks if there is an arrow and attempts to kill the wumpus.

Challenges programming this assignment included the slow execution speed of prover9, the unfamiliarity of prover9 syntax, and the difficulty in designing a working first order logic engine. The python part of the assignment was straight-forward and the A* search was entertaining to program and see working. All issues were resolved by devoting a large amount of time to them. Additionally, the discussion board on canvas was incredibly helpful for clarifying the assignment.