CS-171 Wumpus World Final AI Report

Team name <u>**KekCity**</u>
Member #1 <u>**Geoffrey Ko / 95975359**</u> **Member #2** <u>**Phi Nguyen / 47604490**</u>

I. In about 1/2 page of text, describe what you did to make your Final AI agent "smart."
We implemented a graph to model the cave. Similar to a depth-first search, the AI starts out by proceeding in a single direction until it reaches a dangerous percept (breeze/stench). Upon perceiving a stench, it shoots in the direction it is facing. If a scream is next perceived, then the wumpus has been killed, and the agent can safely ignore all stenches in the cave. Otherwise, it knows that the wumpus is either on its left or right side - it isn't in front of it (or a scream would be perceived), and it isn't behind it (the cell it just came from). Upon perceiving a breeze on vertex *v*, then it will mark *v*'s unexplored neighbors as "potentially dangerous." It will backtrack to the closest unexplored, safe vertex *x*, and explore *x*'s neighbors. If it explores another safe vertex *w* whose neighbor is "potentially dangerous," then the "potentially dangerous" vertex can be marked as safe and unexplored. This same heuristic is applied to marking pits, potential wumpus locations, and the wumpus itself. To navigate from one node to another, we implemented an A* search, with the actual cost *g(n)* being the number of turns it takes to face the destination vertex from the origin vertex, and the heuristic function *h(n)* being the Manhattan Distance from the origin to the destination. We chose the Manhattan Distance as an estimated remaining distance because it does not give the actual cost (i.e., number of turns/moves to get there), but rather a uniform numerical distance. This A* search is used to generally navigate from any given vertex to another, but is most notably apparent in finding a path from the gold back to the start. This was implemented using a min-heap priority queue, ordered by *f(n) = h(n) + g(n)*, following the algorithm learned in lecture. Staying on the cautious side, our agent never dies (at least in our testing). For example, if a breeze is perceived at the starting point, it will immediately exit the cave, rather than risking walking into a pit. If it is exploring the cave, and all possible safe moves have been exhausted before finding the gold, it will return to the start and exit the cave.

II. In about 1/4 page of text, describe problems you encountered and how you solved them.
Some of the problems we ran into were deciding how to represent our map and how to transcribe information about the map in our data structure. We first started by trying to use a 2d array, but we were running into problems correctly representing the grid in a scalable vector. Therefore, we decided to use a graph instead, which makes a lot more sense for a 2d exploration game. We also ran into various problems implementing our algorithms and getting them to work right. One particular problem was handling when we ran into a wall. The way we handled the problem was that after creating a wall node when we bumped into one, we would make our agent recalculate itself back to the original point. This implementation, without us knowing, was quite faulty and sometimes caused deaths without us knowing why. We resolved the problem by simply saving the previous point and backtracking to that point.The final problem we ran into was a strange occurrence where our agent would randomly die when we ran the program on 10000 worlds, but when we individually ran the single world where it died the agent would do fine. This problem hindered our progress greatly because it prevented us from reaching a consistent 200+ score. We later found out what caused the problem was that we saved our maximum axes in a static class variable that is reused by the subsequent worlds when it should not be. We fixed it by defaulting the static variables every time and it added 70 points to our score.

III. In about 1/4 page of text, provide suggestions for improving this project.
The textbook describes the Wumpus World as a knowledge-engineering problem. When we were thinking of ways to approach the project in the early stages, we could not figure out how to concretely implement a FOL knowledge base in code. Perhaps some implementation hints could be touched upon in lecture as a possible strategy of approach. Another inconvenience is the prohibition of creating additional classes to those found in the shell/codebase. When we implemented the project, we separated our entities into individual classes, but had to merge them all back into MyAI for submission purposes. We understand that for the sake of the makefile and running the tournament, it is easiest to restrict additional classes, but this goes against OOP design and makes the code very messy.