Code: Approach 1

```
void intializeValueIteration() {
    Vs = new double[mundo.width][mundo.height];
    for(int i = 1; i < mundo.height; i++) {
        for (int j = 1; j < mundo.width; j++) {
            if (mundo.grid[i][j] == 0) {
                Vs[i][j] = 0;
            }
            else if (mundo.grid[i][j] == 1) {
                Vs[i][j] = 0;
            }
            else if (mundo.grid[i][j] == 2) {
                Vs[i][j] = -1000;
            }
            else {
                Vs[i][j] = 100000;
            }
        }
    }
}

double getValueWithAction(int i, int j, List<Position> neighbors, int action) {
    double sum = 0;
    double incorrectMoveProb = (1 - moveProb)/4;

    for(Position pos : neighbors){
        if(mundo.grid[pos.x][pos.y] == 1 && pos.action == action){
            sum += Vs[i][j] * moveProb;
        }
        else if(mundo.grid[pos.x][pos.y] == 1){
            sum += Vs[i][j] * incorrectMoveProb;
        }
        else if (pos.action == action) {
            sum += Vs[pos.x][pos.y] * moveProb;
        }
        else {
            sum += Vs[pos.x][pos.y] * incorrectMoveProb;
        }
    }
    return sum;
}
```

```
void valueIteration() {
    double delta = 0.001;
    double discount = 0.9;
    double currentChange = 0.0;
    intializeValueIteration();

    do {
        currentChange = 0.0;

        double[][] valueCopy = new double[mundo.width][mundo.height];
        for (int i = 1; i < mundo.height; i++) {
            for (int j = 1; j < mundo.width; j++) {
                if (mundo.grid[i][j] == 1 || mundo.grid[i][j] == 2 || mundo.grid[i][j] == 3) {   //Don't change values for goal and walls
                    valueCopy[i][j] = Vs[i][j];
                    continue;
                }
                List<Position> neighbors = getNeighbors(i, j);
                neighbors.add(new Position(i, j, STAY));
                double sum = 0;

                for (int a = 0; a < 5; a++) {
                    sum = Math.max(sum, getValueWithAction(i, j, neighbors, a));
                }
                double newValue = (discount * sum);

                currentChange = Math.max(currentChange, Math.abs(Vs[i][j] - newValue));

                valueCopy[i][j] = newValue;
            }
        }
        Vs = valueCopy;

    } while(currentChange > delta);
}
```

```
int automaticAction() {
    HashMap<Integer, Double> actions = new HashMap<>();
    actions.put(NORTH, 0.0);
    actions.put(SOUTH, 0.0);
    actions.put(EAST, 0.0);
    actions.put(WEST, 0.0);
    actions.put(STAY, 0.0);

    for(int i = 1; i < mundo.height; i++) {
        for (int j = 1; j < mundo.width; j++) {
            List<Position> neighbors = getNeighbors(i, j);
            double bestValue = Double.MIN_VALUE;
            Position bestAction = new Position(i, j, STAY);

            for(Position pos : neighbors){
                if(Vs[pos.x][pos.y] > bestValue){
                    bestValue = Vs[pos.x][pos.y];
                    bestAction = pos;
                }
            }
            // Sum of best actions based on each square and its probability
            actions.put(bestAction.action, actions.get(bestAction.action) + probs[i][j]);

        }
    }

    // Get highest possible value from probabilities of different actions
    Map.Entry<Integer, Double> maxEntry = null;
    for (Map.Entry<Integer, Double> entry : actions.entrySet()) {
        if (maxEntry == null || entry.getValue()
                .compareTo(maxEntry.getValue()) > 0) {
            maxEntry = entry;
        }
    }

    return maxEntry.getKey();  // default action for now
}
```

Code: Approach 2 –

```
int differentTraversal(){
    Position current = new Position(0,0, STAY);
    double bestValue = Double.MIN_VALUE;
    int bestAction = 0;

    for(int i = 1; i < mundo.height; i++) {
        for (int j = 1; j < mundo.width; j++) {
            if(probs[current.x][current.y] < probs[i][j]){
                current = new Position(i, j,STAY);
            }
        }
    }

    List<Position> neighbors = getNeighbors(current.x, current.y);
    for(Position pos : neighbors){
        if(Vs[pos.x][pos.y] > bestValue){
            bestValue = Vs[pos.x][pos.y];
            bestAction = pos.action;
        }
    }

    return bestAction;
}
```

Analysis:

Version 1 of Iteration value – Values for walls are 0, values for spaces are -1, and stairwells are -10 with goal being 10. This didn't work super well simply because as soon as I got to bigger maps, I would end up with values so small for the spaces in the far corner that they would default to 0. Worked perfectly with mundo_maze, but not with any values that involve stairwells.

Version 2 of Iteration value – Values for walls are still 0, and I modified the algorithm to ignore walls all together as there is no reason to care if we can't even get to that state. Spaces are still -1, stairwells are -10 with goal being 10000. I fixed the issue with Version 1 by increasing the goal state value, and also by decreasing the delta I was checking against (to see the maximum change value for the iteration of the iterationValue function). I changed this from 0.1 to 0.001, and this fixed the issue with some corner squares not having a value and confusing the robot. This worked well, and was able to get the optimal path on the majority of the maps. I noticed, however, that this version of the algorithm threw caution into the wind when it came to being

careful around stairwells. It would always try the optimal path (shortest path) even though it might not guarantee the survival of the robot.

Version 3, final version – I found a few issues with Version 2 and was able to fix them by initializing all space squares to 0, and this helped the robot not get stuck in some weird edge cases, and helped prioritize safety over optimality.

Version 1 of Approach 1 – I found that I got a pretty good algorithm for this approach first try. I initially was just looking at the immediate optimal path, and just used the space that I was most probable to be located in as the judge of where I was. This worked well for higher parameter values where I was pretty sure I was in a certain square, but was not always the best. I would get trapped in stairwells a lot, but the optimal number of moves for the majority of the maps was decently low. This did not do well at all when the initial location was not known.

Version 2, final version – This time I decided that I would take the probability that I was in a given square, and multiply that by the best value and add it to a map that contained all the possible actions. This way, at every given state, I could get the best possible action given all the probabilities of my location. This took into account uncertainty, and just gave me an average of all the best actions given the probability that am in any of my believed squares. This worked really well, and not only helped prioritize security, but was able to find a quick path rather frequently.

Approach 2:

For this function, I simply pick the most probable square and make the optimal move for it (move to the square with the most value). This works well for higher parameter values, but when the values are low, it does not work at all, and results in a very confused robot.

Iteration Value Variables:

Discount = 0.9
Delta = 0.001
Reward = 0

I found that the algorithm worked best with these parameters. I wanted to keep the discount high so that I wouldn't end up with really small decimals in the bigger maps. I also kept the delta low so that I could have a more accurate value map. The reward was simply easier to keep at 0, and helped the iteration value better find optimal values.