1. **start(3, 3).**: This defines the starting point of the maze as the coordinates **(3, 3)**.
2. **goal(1, 1).**: This defines the goal point of the maze as the coordinates **(1, 1)**.
3. **blocked(2, 2).**: This defines a blocked square in the maze at the coordinates **(2, 2)**.
4. **blocked(2, 3).**, **blocked(2, 4).**, **blocked(3, 2).**, **blocked(3, 4).**, **blocked(4, 2).**: These are additional blocked squares in the maze.
5. **move(up, X/Y, X/Y1) :- Y > 1, Y1 is Y-1, \+ blocked(X,Y1).**: This defines a movement rule for moving up one square. The predicate **move/3** takes three arguments: the direction of movement (in this case, **up**), the current coordinates (**X/Y**), and the resulting coordinates (**X/Y1**). The rule first checks that moving up is a valid move (i.e., the current Y coordinate is greater than 1 and the square above is not blocked). If it is a valid move, it sets **Y1** to the new Y coordinate (i.e., one square up from the current position).
6. **move(right, X/Y, X1/Y) :- X < 5, X1 is X+1, \+ blocked(X1,Y).**: This defines a movement rule for moving right one square. The rule first checks that moving right is a valid move (i.e., the current X coordinate is less than 5 and the square to the right is not blocked). If it is a valid move, it sets **X1** to the new X coordinate (i.e., one square to the right from the current position).
7. **move(down, X/Y, X/Y1) :- Y < 5, Y1 is Y+1, \+ blocked(X,Y1).**: This defines a movement rule for moving down one square. The rule first checks that moving down is a valid move (i.e., the current Y coordinate is less than 5 and the square below is not blocked). If it is a valid move, it sets **Y1** to the new Y coordinate (i.e., one square down from the current position).
8. **move(left, X/Y, X1/Y) :- X > 1, X1 is X-1, \+ blocked(X1,Y).**: This defines a movement rule for moving left one square. The rule first checks that moving left is a valid move (i.e., the current X coordinate is greater than 1 and the square to the left is not blocked). If it is a valid move, it sets **X1** to the new X coordinate (i.e., one square to the left from the current position).
9. **dfs(Goal, Path) :- start(StartX, StartY), dfs\_helper(Goal, [(StartX, StartY)], Path).**: This defines the depth-first search algorithm. The **dfs/2** predicate takes two arguments: the goal point of the maze (**Goal**) and a variable that will be bound to the resulting path (**Path**). The predicate first gets the starting point of the maze (**StartX** and **StartY**) and calls the **dfs\_helper** predicate with the goal point and a list containing the starting point.
10. **dfs\_helper(Goal, [(GoalX, GoalY)|Visited], [(GoalX, GoalY)|Visited]) :- Goal = (GoalX, GoalY).**: This is the base case of the **dfs\_helper** predicate. It checks if the current location of the search (**(GoalX, GoalY)**) is the goal location (**Goal**). If it is, then it simply returns the current path, which is the list of visited locations.
11. **dfs\_helper(Goal, [(X, Y)|Visited], Path) :- \+ Goal = (X, Y), move(\_, X/Y, X1/Y1), \+ member((X1, Y1), [(X, Y)|Visited]), dfs\_helper(Goal, [(X1, Y1),(X, Y)|Visited], Path).**: This is the recursive case of the **dfs\_helper** predicate. It checks if the current location of the search (**(X, Y)**) is not the goal location (**\+ Goal = (X, Y)**). It then tries to move in any direction using the **move/3** predicate (using the anonymous variable **\_** to indicate that the direction is not relevant). If the resulting location (**(X1, Y1)**) is not in the list of visited locations (**\+ member((X1, Y1), [(X, Y)|Visited])**), it is added to the list of visited locations (**[(X1, Y1),(X, Y)|Visited]**) and the **dfs\_helper** predicate is called recursively with the updated list of visited locations.
12. **ids(Goal, Path) :- start(StartX, StartY), ids\_helper(Goal, [(StartX, StartY)], Path, 0).**: This defines the iterative deepening search algorithm. The **ids/2** predicate takes two arguments: the goal point of the maze (**Goal**) and a variable that will be bound to the resulting path (**Path**). The predicate first gets the starting point of the maze (**StartX** and **StartY**) and calls the **ids\_helper** predicate with the goal point, a list containing the starting point, and a depth limit of 0.
13. **ids\_helper(Goal, [(GoalX, GoalY)|Visited], [(GoalX, GoalY)|Visited], \_) :- Goal = (GoalX, GoalY).**: This is the base case of the **ids\_helper** predicate. It checks if the current location of the search (**(GoalX, GoalY)**) is the goal location (**Goal**). If it is, then it simply returns the current path, which is the list of visited locations.
14. **ids\_helper(Goal, [(X, Y)|Visited], Path, Depth) :- \+ Goal = (X, Y), Depth < 10, move(\_, X/Y, X1/Y1), \+ member((X1, Y1), [(X, Y)|Visited]), Depth1 is Depth+1, ids\_helper(Goal, [(X1, Y1),(X, Y)|Visited], Path, Depth1).**: This is the recursive case of the **ids\_helper** predicate. It checks if the current location of the search (**(X, Y)**) is not the goal location (**\+ Goal = (X, Y)**) and the current depth (**Depth**) is less than a maximum depth limit (10 in this case). It then tries to move in any direction using the **move/3** predicate (using the anonymous variable **\_** to indicate that the direction is not relevant). If the resulting location (**(X1, Y1)**) is not in the list of visited locations (**\+ member((X1, Y1), [(X, Y)|Visited])**), it is added to the list of visited locations (**[(X1, Y1),(X, Y)|Visited]**) and the **ids\_helper** predicate is called recursively with the updated list of visited locations and an incremented depth limit (**Depth1 is Depth+1**). If the goal is not found at the current depth limit, the **ids\_helper** predicate will be called again with a higher depth limit until either the goal is found or the maximum depth limit is reached. The final result will be the path to the goal with the fewest number of steps.

Note that the maximum depth limit is set to 10 in this implementation, but you can change it to any value that suits your needs.

Also, note that the **move/3** predicate defines the rules for moving in each direction (up, down, left, and right) in the maze. It checks if the move is valid (i.e., if the resulting location is within the bounds of the maze and is not blocked) and returns the new location if the move is valid.

Finally, note that the **blocked/2** predicate defines the locations that are blocked in the maze (i.e., the locations that cannot be visited during the search)