Problem Description

An AI agent, which carries an egg, has to go into a randomly generated maze and fetch randomly placed pokemons and spend enough time in the maze for the egg to hatch.

1. **Environment:**

The agent knows its environment; as it knows the pokemons' locations and number, the maze structure, and the initial , and the end cell in the maze.

2. State Definition:

The state in this problem is defined as:

1. Agent's current cell coordinates
2. The number of collected pokemons so far
3. The time spent so far
4. The situation in which the action was applied

3. The Operators:

The agent can only perform 3 actions which are:

1. Move forward
2. Move left
3. Move right
4. Move backward

4. The Goal Test:

A state can be considered a goal test if and only if:

1. The egg hatches
2. All the pokemons in the maze are collected

Knowledge Base Generation

The maze configurations were generated in java and then written into a pl file to be loaded as the knowledge base for the agent and the generated constants were as follow:

1. Initial Cell:
   at(X,Y,0,0): this indicates that initially the agent is at cell(X,Y) and has P pokemons to collect and T time for the egg to hatch.

2. Pokemons Location:
   hasPokemon(X,Y): denotes that there is a pokemon at cell(X,Y).

3. Cell Openings:
   cellOpen(X,Y,Orientation): denotes that cell(X,Y) is open or has an entry point from the Orientation side.

4. End Cell:
   query(X,Y,P,T,S): this is the query that would be used to get the sequence of actions S to collect the pokemons and reach the end cell.

5. Maze Size:
   width(X), height(X): denotes the maze dimensions.

6. nonvar(X):

   predefined predicate in prolog which returns true if X is not a variable.

7. number(X):

   predefined predicate in prolog which returns true if the X is a number.

Java Method:

- void GenerateKB(): this method initializes the maze and scan each cell to get its opened walls and to locate the pokemons. Also it generates the initial and end cell predicates and maze size predicates.

Successor State Axioms

Fluent Description:

There are two fluent in this problem. In other words, states or conditions that change by time. These fluent have an extra situation argument. In this problem the only two things that change are:

1. The agent's position
2. The pokemon's location (collected pokemons are removed from their cells)

Hence, the At and hasPokemon predicate would have an extra argument S that denotes the situation in which these predicates hold. Consequently, initially the generated KB would generate only the maze configuration in the initial situation (S0) in the following manner:

1. initial cell: at(X,Y,0,0,S0)
2. pokemons location: hasPokemon(X,Y,S0)

Other predicates would remain the same as they are static. For example, a cell would be open from the right at any given situation.

Successor State Axioms:

1. **The at predicate:**

   the predicate at(x, y, p, t, Result(a, s)) would hold if and only if the applied action is either:

   1. *forward* and the agent was in the cell below the destination cell and its wall is open from upwards
   2. *backward* and the agent was in the cell above the destination cell and its wall is open from downwards
   3. *right* and the agent was in the cell on the right of the destination cell and its wall is open from the right
   4. *left* and the agent was in the cell on the left of the destination cell and its wall is open from the left

   the effect of these actions is:

   - increment the time spent in the maze
   - if the cell has a pokemon, the pokemon counter is incremented

Or the agent could have been at cell (x, y) and the action that was applied failed. Any of these action would fail if and only if the cell the agent moves to is not open from the agent's entry point. For example the agent would move forward and the destination cell is not open from upwards.

$$\forall x, y, p, t, s, a\ [\ at(x, y, p, t, Result(a, s)) \iff$$

$$[\ (a = f\ \wedge\ cellOpen(forward)\ \wedge\ \exists\ x1, p1, t1[\ at(x1, y, p1, t1, s)\ \wedge\ (x = x1 - 1)\ \wedge\ (t = t1 + 1)\ \wedge\ hasPokemon(x, y, s) \rightarrow (p = p1 + 1)])$$

$$\vee$$

$$(a = b\ \wedge\ cellOpen(back)\ \wedge\ \exists\ x1, p1, t1[\ at(x1, y, p1, t1, s)\ \wedge\ (x = x1 + 1)\ \wedge\ (t = t1 + 1)\ \wedge\ hasPokemon(x, y, s) \rightarrow (p = p1 + 1)])$$

$$\vee$$

$$(a = r\ \wedge\ cellOpen(right)\ \wedge\ \exists\ y1, p1, t1[\ at(x, y1, p1, t1, s)\ \wedge\ (y = y1 - 1)\ \wedge\ (t = t1 + 1)\ \wedge\ hasPokemon(x, y, s) \rightarrow (p = p1 + 1)])$$

$$\vee$$

$$(a = l\ \wedge\ cellOpen(left)\ \wedge\ \exists\ y1, p1, t1[at(x, y1, p1, t1, s)\ \wedge\ (y = y1 + 1)\ \wedge\ (t = t1 + 1)\ \wedge\ hasPokemon(x, y, s) \rightarrow (p = p1 + 1)])]]$$

$$\vee$$

$$[at\ (x, y, p, t, s)$$

$$\wedge$$

$$(a = f\ \wedge\ \neg(cellOpen(forward)))\ \vee\ (a = b\ \wedge\ \neg(cellOpen(back)))$$

$$\vee\ (a = r\ \wedge\ \neg(cellOpen(right)))\ \vee\ (a = l\ \wedge\ \neg(cellOpen(left)))]$$

2. hasPokemon predicate:

the predicate hasPokemon(x , y, Result(a, s)) would hold if and only if the agent is not moving into a cell that already has pokemon; as the agent would collect the pokemon if it entered the cell (refer to the at predicate).

$$\forall x,y,a,s\ [\text{hasPokemon}(x, y, \text{Result}(a, s)) \iff$$

$$\varphi\ v$$

$$[\ \text{hasPokemon}(x, y, s)$$

$$\wedge$$

$$[(\ \neg(a = f) \wedge\ \exists x1, p, t[\ \text{at}(x1, y, p, t, s) \wedge (x1 = x + 1)])$$

$$v$$

$$(\ \neg(a = b) \wedge\ \exists x1, p, t[\ \text{at}(x1, y, p, t, s) \wedge (x1 = x - 1)])]$$

$$v$$

$$(\ \neg(a = r) \wedge\ \exists y1, p, t[\ \text{at}(x, y1, p, t, s) \wedge (y1 = y - 1)])]$$

$$v$$

$$(\ \neg(a = l) \wedge\ \exists y1, p, t[\ \text{at}(x, y1, p, t, s) \wedge (y1 = y + 1)])]]$$

Plan Generation

The knowledge base is queried by calling the query predicate

query(X,Y,Po,T,P):-

query1(X,Y,Po,T,P,4) predicate will be called

* Note:- the 6th parameter in query1 represents the depth limit which we will later use in call_with_depth_limit predicate. We chose it to be 4 not 1 because in the simplest maze the agent had to make at least 4 steps to reach a goal state. And it made our life easy in tracing.

query1(X,Y,Po,T,P,C):-

X: X coordinate of the agent location.

Y: Y coordinate of the agent location.

Po: Number of Pokemons collected so far.

T: remaining time to hatch the egg.

P: sequence of steps the agent made to reach the goal.

C: depth limit which is used in calling call_with_depth_limit predefined predicate.

This predicate handles 2 cases depending on the value of R:

1. the search will find a solution within the given depth
2. the search didn't find a solution within that depth and it returns depth_limit_exceeded.

In this predicate findSolution(X, Y, Pokemons, TimeHatch, P, C, R) is called

And checks on the result R:

- If the R value is equal to depth_limit_exceeded this means the agent didn't find a solution within the given depth so the depth limit is increment and query1 is called again recursively.

- If the R value is not equal to depth_limit_exceeded this means the agent found a solution within the given depth and no need to make a recursive call.

findSolution(X,Y,Pokemons,TimeHatch,P,Limit,Result):-

It calls the call_with_depth_limit predefined predicate.

The need for call_with_depth_limit predicate

Prolog uses DFS search to obtain a solution. Since DFS is not complete, a solution is not always guaranteed. Consequently, call_with_depth_limit predefined predicated is called with the at predicate as a query and a depth limit that is implemented recursively, simulating the iterative deepening strategy discussed in the lecture. Since iterative deepening is a complete search strategy, the generated plan is also complete.

call_with_depth_limit(at(X,Y,P,T,S), Limit, R):-

call with depth limit calls the at predicate which in turn calls itself recursively and calls hasPokemon predicate, which also calls itself recursively as illustrated in Fig.1 below.

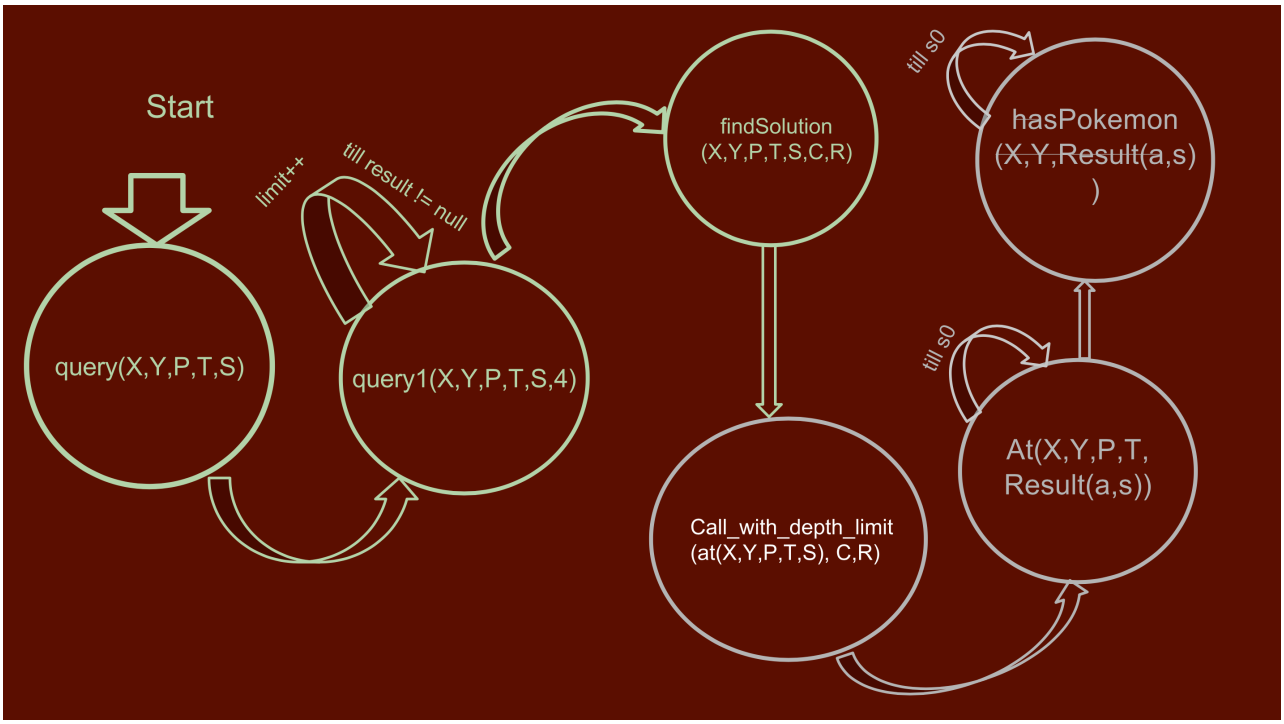


Fig.1: Plan generation map

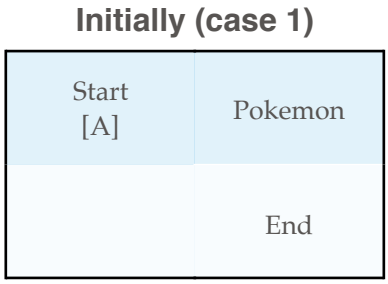
Code Output

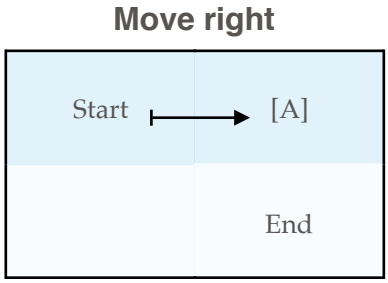
Case 1:

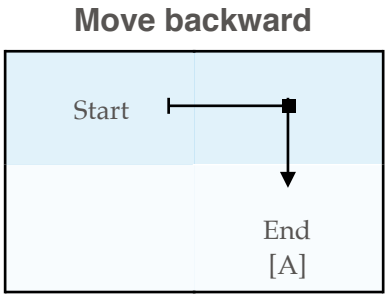
Sequence of actions:

```
[?- query(1,1,1,1,P).
P = result(b, result(r, s0)) .
```

Fig.2: Prolog output (Case 1)







Case 2:

Sequence of actions:

```
[?- query(0,0,1,1,P).
P = result(l, result(f, s0)) .
```

Fig.3: Prolog Output (Case 2)