Artificial Intelligence Assignment1

Marina Smirnova Group 6

March 2020

Contents

1 Introduction

1.1 Execution and Launching

The submitted archive contains a report, a file with source code and a folder with input maps for testing.

I would strongly appreciate if you execute the program via SWI-Prolog IDE (in case of testing). The reason is that there are several predicates that are built in this IDE, but not provided by online compilers.

At the very top of the source code there are some commented lines that allows to download the Knowledge Base for a certain map. All maps are in the folder Input_Maps, and are specified in the Tests Section. So, for testing a map, you just need to delete % sign before the line. And before starting another test comment it again.

There are 2 predicates to launch the game: *writeResultsandTime/1* and *startMatch/1*. The first one prints time for execution and results(min counter and min path). The second one prints only results. A single argument that they accept is Algorithm(1-Random, 2-Backtracking, 3-Optimized).

2 Preview

2.1 General assumptions about the rules of the game(environment)

List of assumptions(additional ones):
• Environment:

1. The environment is a square field 20 by 20. But for simplicity it's assumed to be 5 by 5(or at most 10 by 10),and all examples of maps have such size.
2. It's assumed that there is definitely at least 1 orc/human/touchdown on the map.
3. The min path and min counter are found for the first touchdown stated in the Knowledge Base(if it's possible, otherwise next fact with touchdown location is tried).

• Player:

1. A successful pass to another human is considered to be 1 move.
2. A turn is not considered to be a move.

• Random Algorithm:

1. The main principle of the algorithm is to choose direction of motion and pass randomly. And at any moment of the game, there is only 1 choice possible - no other alternatives are considered.

• Backtracking Algorithm:

1. This algorithm checks all 4 direction of motion and all 8 directions of pass at each cell of the field.

• Optimized Backtracking Algorithm:

1. The algorithm is supposed to have heuristics behind of its decision making. However, it's not allowed to use any information about touchdown location. So, it was decided to not use any heuristics with weights for any cell(as in Manhattan A*). And a simple constraint "do not go further, if you already exceed number of moves found previously" was used.

3 Implementation

3.1 Basic functionality

This subsection contains several rules that are common for all three algorithms.

•Initial Knowledge Base and dynamic predicates:
:-dynamic([minCounter/1, minPath/1, pass/1]).
There 3 dynamic predicates to hold "global" min counter, min path (as a list) and pass status (1-allowed, 0-already done).

The game starts by using *writeResultAndTime(Algorithm)* or *startMatch(Algorithm)*. The first one will print result of the game and time in milliseconds taken by execution. The second one only the result. The variable Algorithm: 1-Random Search, 2-Backtracking, 3-Optimized Backtracking. The actual predicate that chooses the algorithm and launch it is *launchAgent/1*.

There is also a rule for initialization of dynamic facts: *initGame/0*. It sets min path to an empty list, pass status to 1, min counter to 1000.

•Predicates connected with location:

```
%----getLocation----
getXYofLocation(Location, X, Y):-
    Location = [H1|T],
    X is H1,
    T = [H2|_],
    Y is H2.
```

All cells have coordinates X and Y represented as two-valued lists [X, Y]. And this predicate extracts X and Y from Location list.

```
getRightCell(Location,NewLocation):-
     getXYofLocation(Location,X,Y),
     NewX is X+1, NewLocation = [NewX,Y].
getLeftCell(Location,NewLocation):-
     getXYofLocation(Location,X,Y),
     NewX is X-1, NewLocation = [NewX,Y].
getUpCell(Location,NewLocation):-
     getXYofLocation(Location,X,Y),
     NewY is Y+1, NewLocation = [X,NewY].
getDownCell(Location,NewLocation):-
     getXYofLocation(Location,X,Y),
     NewY is Y-1, NewLocation = [X,NewY].
```

These predicates allow to get the next cell in the needed direction. There are rules:

updateLocation(Location,Direction,NewLocation) and *lookAt(Location,Direction,Path,NextCellState)* that use given predicates.

updateLocation(Location,Direction,NewLocation) : instantiates NewLocation variable with computed [X,Y] in accordance with Direction and current Location.

lookAt(Location,Direction,Path,NextCellState) : given the Direction as a number [1..4] and a Location as a tuple [X,Y], it gets needed cell and determines its state. Directions: 1-right, 2-left, 3-up, 4-down. This predicate also calls a rule (detectObject/3) to determine the type of object on the given cell if any.

```
detectObject(Location,Path,CellState):-
    (orcDetected(Location), CellState is 1,!);
    (playerDetected(Location), CellState is 2,!);
    (touchdownDetected(Location), CellState is 3,!);
    (visitedLocation(Location,Path), CellState is 5,!);
    CellState is 0.
```

detectObject(Location,Path,CellState) : instantiates CellState variable with apropriate value. Cuts are used to prevent incorrect backtracking and assigning 0 as a cell's state.

theEnd(Location,Path,MoveCounter,EndCounter) : this predicate determines whether a player reached the touchdown cell. And it updates dynamic predicates minCounter and minPath, if so and if found MoveCounter and Path are better than previously found min values. And EndCounter variable returns the currently found min value of MoveCounter to pass to *aggregate_all/4* predicate.

•**Output:**

```
printResults:-
    (
    minPath(Path),Path \== [],
    write("Min counter "),
    minCounter(Min),writeln(Min),
    writeln("Min path "),
    printReversedPath(Path)
    );
    (writeln("Game Over. No path. Or no correct random path.")).

printReversedPath(Path):- reverse(Path,_,ReversedPath),printPath(Reversed
Path).

printPath([]):- !.
printPath([H|T]):-
    writeln(H), printPath(T).

reverse([],Buffer,Buffer).
reverse([H|T],Buffer,ReversedList):-
    reverse(T,[H|Buffer],ReversedList).
```

To print path we, firstly, need to reverse the order of locations in it, because they are appended at beginning of the list.

•Update of variables:
updateMinCounter(MoveCounter) : updates dynamic predicate minCounter/1.
updateMinPath(Path) : updates dynamic predicate minPath/1.
updateMoveCounter(MoveCounter,CellState,NewMoveCounter) : assigns a value to NewMoveCounter variable. It is either (MoveCounter +1) or just MoveCounter, if free pass to another human on the current cell is done.
updatePath(NewLocation,Path,NewPath) : assigns Path with appended Location to a NewPath variable.
updateLocation(Location,Direction,NewLocation) : assigns new coordinates to a NewLocation variable in accordance with the direction of motion.
updateParameters(Location,Direction,Path,NewLocation,NewPath) : calls *updatePath/3* and *updateLocation/3*.

3.2 Random Search

Firstly, the algorithm for Random Search should be launched. The idea is to execute the predicate *startRandomSearch/4* for 100 times.

```
%----Random----
executeRandomSearch:-
    (initGame,
    executeRandomNTimes(100));
    printResults.

executeRandomNTimes(Count) :-
    Count > 0,
    (
        (
        format('----------------~n'),

        retractall(pass(_)),
        assert(pass(1)),

        (startOnPlayer, MoveCounter is -1; MoveCounter is 0),

        startRandomSearch([0,0], [[0,0]], MoveCounter,_)
        );

        (NewCount is Count - 1,
        executeRandomNTimes(NewCount))
    ).
```

The thing to note: only if a human gets to touchdown location the result with min counter and min path will be printed. Otherwise, in case of death, bumping into the wall or unsuccessful pass the attempt(one out of 100) is failed and another attempt is started. And status of pass is set to be allowed (value 1) for each new attempt.

The main predicate that performs Random Search.

```
startRandomSearch(AgentLocation,Path,MoveCounter,EndCounter):-
    (theEnd(AgentLocation,Path,MoveCounter,EndCounter),writeln("Win."),!,true);

    (orcDetected(AgentLocation),writeln("Death. Game Over."),!,fail);

    (movesLimit(MoveCounter),writeln("No more moves. Game Over."),!,fail);

    (wallDetected(AgentLocation), write(AgentLocation), writeln("Bumping. Game Over."), !,fail);

    (
        not(wallDetected(AgentLocation)),

        (determineRandomAction(AgentLocation,Path,MoveCounter,NewLocation,NewPath,NewMoveCounter),

        startRandomSearch(NewLocation,NewPath,NewMoveCounter,EndCounter)
        )
    ).


determineRandomAction(Location,Path,MoveCounter,NewLocation,NewPath,NewMoveCounter):-
    chooseActionRandomly(Action),
    (
        (Action = 0,
         changeDirectionRandomly(NewDirection),
         lookAt(Location,NewDirection,Path,NextCellState),
         updateMoveCounter(MoveCounter,NextCellState,NewMoveCounter),
         updateParameters(Location,NewDirection,Path,NewLocation,NewPath));

        (Action = 1,
         makeRandomPass(Location,MoveCounter,NewLocation,NewMoveCounter),
         updatePathForPass(Path,NewPath)
        )
    ).


% 0 move; 1 pass
chooseActionRandomly(Action):-
    pass(Permission),
    (
        (Permission = 1, random_between(0,1,Action));
        (Permission = 0,Action is 0)
    ).
```

Used predicates:

movesLimit/1 : checks wether the player has done 100 moves. If yes, then quit the attempt. Otherwise continue.

determineRandomAction/6 : chooses between a pass or a move. And Action = 0 is for motion, Action = 1 is for pass.

changeDirectionRandomly/1 : returns a randomly chosen direction (1-right, 2-left, 3-up, 4-down).

makeRandomPass/4 :

```
makeRandomPass(Location,MoveCounter,NewLocation,NewMoveCounter):-
    updatePass,

    chooseRandomPassDirection(PassDirection),
    checkSuccessOfPass(Location,PassDirection,Success,NewLocation),
    (
        (Success = 1,NewMoveCounter is MoveCounter+1);
        (Success = 0,writeln("Wasted Pass. Game Over."),fail)
    ).
```

Note: there are no other attempts to choose another direction of motion or pass (no alternatives must be considered, that is why cuts are used in chooseRandomPassDirection/1 and changeDirectionRandomly/1).

checkSuccessOfPass(Location,Direction,Success,PossibleLocation) : gets the randomly chosen direction and executes the appropriate predicate to check success of the pass in this direction. PossibleLocation variable will hold the location of a human who can catch the ball (if pass is possible).

There are 8 predicates for each possible direction of pass, and each differs from one another by different arithmetic calculations for coordinates of the trajectory.

```
%----RightPass----
checkRTrajectory(X,Y,Success,PossibleLocation):-
    maxX(MaxX),
    (
        X =< MaxX,
        (
         (orcDetected([X,Y]), Success is 0,!);
         (playerDetected([X,Y]), Success is 1, PossibleLocation=[X,Y],!);
         (NewX is X+1,checkRTrajectory(NewX,Y,Success,PossibleLocation))
        )
    ).
```

3.3 Backtracking Search

To find the best min path and counter, it is needed to consider all possible paths. Therefore, the predicate *aggregate_all/4* is used. It finds the min possible value of returned variable EndCounter
The main predicate that performs Backtrack Search.

Used predicates:

determineAction/6 : actually this predicate is responsible for motion. It checks safety of each cell in the sight area of the player(right, left, up, down) and updates parameters if the move is possible.

makePass/3 : is similar to the predicate checkSuccessOfPass/4. But it does not need a variable Direction, because all 8 directions are considered one by one.

3.4 Heuristic Search

The heuristics added to reduce time of search is the condition: compare current MoveCounter with "global" minCounter, and if the current one is greater, than there is no sense to proceed further on this path.

The main predicate that performs Optimized Search.

```
%----Backtacking----
startBacktrackingSearch:-
    (startOnPlayer, MoveCounter is -1; MoveCounter is 0),
    aggregate_all(min(EndCounter),startBacktracking([0,0],[[0,0]],1,MoveCounter,EndCounter),EndCounter),
    printResults,!,true.


startBacktracking(AgentLocation,Path,PassStatus,MoveCounter,EndCounter):-
    (theEnd(AgentLocation,Path,MoveCounter,EndCounter),!,true);

    (
        not(wallDetected(AgentLocation)),

        (
            (
               determineAction(AgentLocation,Path,MoveCounter,NewLocation,NewPath,NewMoveCounter),
               startBacktracking(NewLocation,NewPath,PassStatus,NewMoveCounter,EndCounter)
            );

            (
             PassStatus = 1, NewPassStatus is 0,
             makePass(AgentLocation,Success,PossibleLocation),
             Success = 1,
              NewMoveCounter is MoveCounter+1,
             updatePathForPass(Path,NewPath),
             startBacktracking(PossibleLocation,NewPath,NewPassStatus,NewMoveCounter,EndCounter)
            )

        )
    );

    (minPath(MP), MP = [], EndCounter is 0).



%----DetermineAction----
determineAction(Location,Path,MoveCounter,NewLocation,NewPath,NewMoveCounter):-
    (
        %writeln("try move right"),
        lookAt(Location,1,Path,NextCellState),
        isSafe(NextCellState),
        updateMoveCounter(MoveCounter,NextCellState,NewMoveCounter),
        updateParameters(Location,1,Path,NewLocation,NewPath)
    );
    (
        %writeln("try move left"),
        lookAt(Location,2,Path,NextCellState),
        isSafe(NextCellState),
        updateMoveCounter(MoveCounter,NextCellState,NewMoveCounter),
        updateParameters(Location,2,Path,NewLocation,NewPath)

    );
    (
        %writeln("try move up"),
        lookAt(Location,3,Path,NextCellState),
        isSafe(NextCellState),
        updateMoveCounter(MoveCounter,NextCellState,NewMoveCounter),
        updateParameters(Location,3,Path,NewLocation,NewPath)

    );
    (
        %writeln("try move down"),
        lookAt(Location,4,Path,NextCellState),
        isSafe(NextCellState),
        updateMoveCounter(MoveCounter,NextCellState,NewMoveCounter),
        updateParameters(Location,4,Path,NewLocation,NewPath)

    ).
```

```
%----OptimizedBacktracking----
startOptimizedSearch:-
    (startOnPlayer, MoveCounter is -1; MoveCounter is 0),
    aggregate_all(min(EndCounter),startOptimized([0,0],[[0,0]],1,MoveCounter,EndCounter),EndCounter),
    printResults,!,true.


startOptimized(AgentLocation,Path,PassStatus,MoveCounter,EndCounter):-
    (theEnd(AgentLocation,Path,MoveCounter,EndCounter),!,true);

    (
        not(wallDetected(AgentLocation)),
        continueSearch(MoveCounter),

        (
            (
               determineAction(AgentLocation,Path,MoveCounter,NewLocation,NewPath,NewMoveCounter),
               startOptimized(NewLocation,NewPath,PassStatus,NewMoveCounter,EndCounter)
               );

            (
             PassStatus = 1, NewPassStatus is 0,
             makePass(AgentLocation,Success,PossibleLocation),
             Success = 1,
             NewMoveCounter is MoveCounter+1,
             updatePathForPass(Path,NewPath),
             startOptimized(PossibleLocation,NewPath,NewPassStatus,NewMoveCounter,EndCounter)
             )

        )
    );

    (minPath(MP), MP = [], EndCounter is 0).


continueSearch(MoveCounter):-
    minCounter(Min), MoveCounter < Min.
```

4 Tests

4.1 Corner cases

Problematic maps and solutions for them:

- An orc / a touchdown on the start [0,0]

```
startMatch(Algorithm):-
    initGame,
    (
        (not(startOnOrc),not(startOnTouchdown),launchAgent(Algorithm));
        (startOnOrc, writeln("Impossible to move: orc at [0,0]."));
        (startOnTouchdown, writeln("Win. Touchdown on [0,0]"))
    ).
```

```
startOnOrc:-
    o(OrcLocation),
    isEqual([0,0],OrcLocatio
```

```
startOnTouchdown:-
    t(TouchdownLocation),
    isEqual([0,0],TouchdownLocation).
```

Import files CornerCase1.pl or CornerCase2.pl with Knowledge Bases for testing.

- A player / a touchdown is surrounded by orcs

As a solution, I used a comparison between dynamic variable minPath (like a global variable denoting min possible path to the touchdown) and empty list. It allows me to note whether the player got to the touchdown, because only then the minPath is modified from empty list to currently found path.

Import files CornerCase3.pl or CornerCase4.pl with Knowledge Bases for testing.

4.2 Small Maps 5 by 5

Here are several examples of maps, that can be found in the folder Input_Maps. Their names are MapN_5x5.pl. They check the functionality fully, and the program will work in the same correct way if tested with a bigger map.

- **Map1**

The main purpose: check pass functionality.
Output results:
Min counter = 3
Min path
[0, 0] pass [3, 4] [4, 4]

- **Map2**

The main purpose: check motion within constraints.
Output results:
Min counter = 7
Min path
$[0,0]$ $[0,1]$ $[1,1]$ $[2,2]$ $[2,3]$ $[3,3]$ $[3,4]$

- **Map3**

The main purpose: check pass functionality from the right cell and in the different direction.
Output results:
Min counter = 5
Min path
$[0,0]$ $[1,0]$ $[2,0]$ $[3,0]$ pass $[0,4]$

- **Map4**

The main purpose: check whether pass is made to to the right human.
Output results:
Min counter = 3
Min path
$[0,0]$ $[1,0]$ pass $[4,4]$

- **Map5**

The main purpose: check decision making between pass and turnover (throw the ball or give it to the human - what is more efficient in terms of move counter).
Output results:
Min counter = 2
Min path
$[0,0]$ pass $[0,4]$

- **Map6**

The main purpose: check whether a pass can be caught by an orc.Unsolvable map.
Output results:
Game Over. No path. Or no correct random path.

4.3 Big Map 10 by 10

- **Map1**

Output results:
Min counter = 8
Min path
$[0,0]$ $[1,0]$ $[2,0]$ $[2,1]$ $[2,2]$ $[2,3]$ $[2,4]$ $[2,5]$ $[2,6]$

(a) Map1

				T
			H	O
P				

(b) Map2

H		O	T	O
	O			O
O			O	
		O		
P	O			

(a) Map3

T				
H				
O				
P				

(b) Map4

				T
				H
	H			
P				O

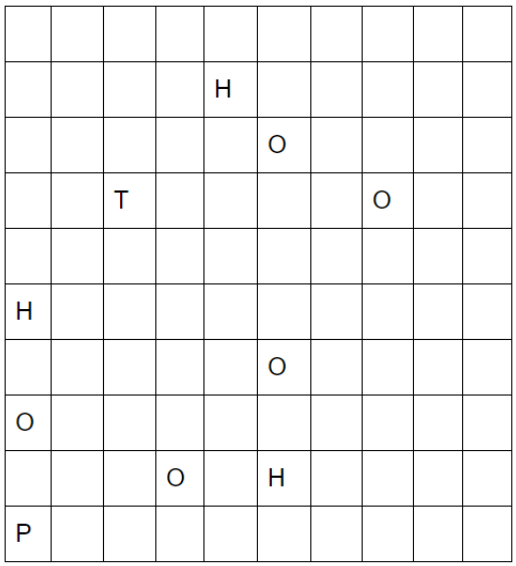
(a) Map5

T				
H				
P				O

(b) Map6

				H
			O	T
O				
P	O			

Figure 5: Big Map 10x10



5 Analysis

I would like to make a conclusion about efficiency of all three algorithms using data collected through different testing maps. The table below contains time results in milliseconds for each map (described above) and algorithm.

Figure 6: Table with time results for tested maps

Algorithm/Time, ms	Random Search	Backtracking	Optimized
Map1_5x5	found, 45096	29728	22
Map2_5x5	failed	16	16
Map3_5x5	found, 23050	370921	14
Map4_5x5	found, 8564	309054	25
Map5_5x5	failed	605677	11
Map6_5x5	failed	0	0
Map1_10x10	failed	15 min	2209

It's clear that the less time-consuming algorithm is Optimized Search. The Backtracking is good due to its robustness in comparison with Random Search. The last one does not guarantee finding any solution at all, because it easily terminates due to player's death, bumping or unsuccessful pass.

6 Additional questions to consider

Vision of 2 cells
If we consider the condition of seeing 2 cells in the given four directions, the results produced by Backtracking and Optimized searches will change. The Random Algorithm will not be affected, because it does not use the information about the neighbourhood to make the next step.

In both Backtracking and Optimized there are a rule that is responsible for making moving decisions, a rule for discovering the adjacent cells a rule to distinguish between encountered objects(a human, an orc, a touchdown, a wall) and a rule to determine safety of the next possible cell. These predicates: *determineAction/6*, *lookAt/4*, *detectObject/3*, *isSafe/1*.

The predicate *detectObject(Location,Path,CellState)* returns the state(object type) detected on the cell with coordinates Location. Then this CellState is used in *isSafe(CellState)*.

So, if I tune the *detectObject/3* to discover area within 2 cells instead of 1, I can actually improve optimality of the algorithms.

For instance, consider the case when touchdown is located within 2 cells neighbourhood, and is detected, then the win is achieved. But with only one cell vision the algorithm may choose another direction and, therefore, go off the right path and spend more time searching.