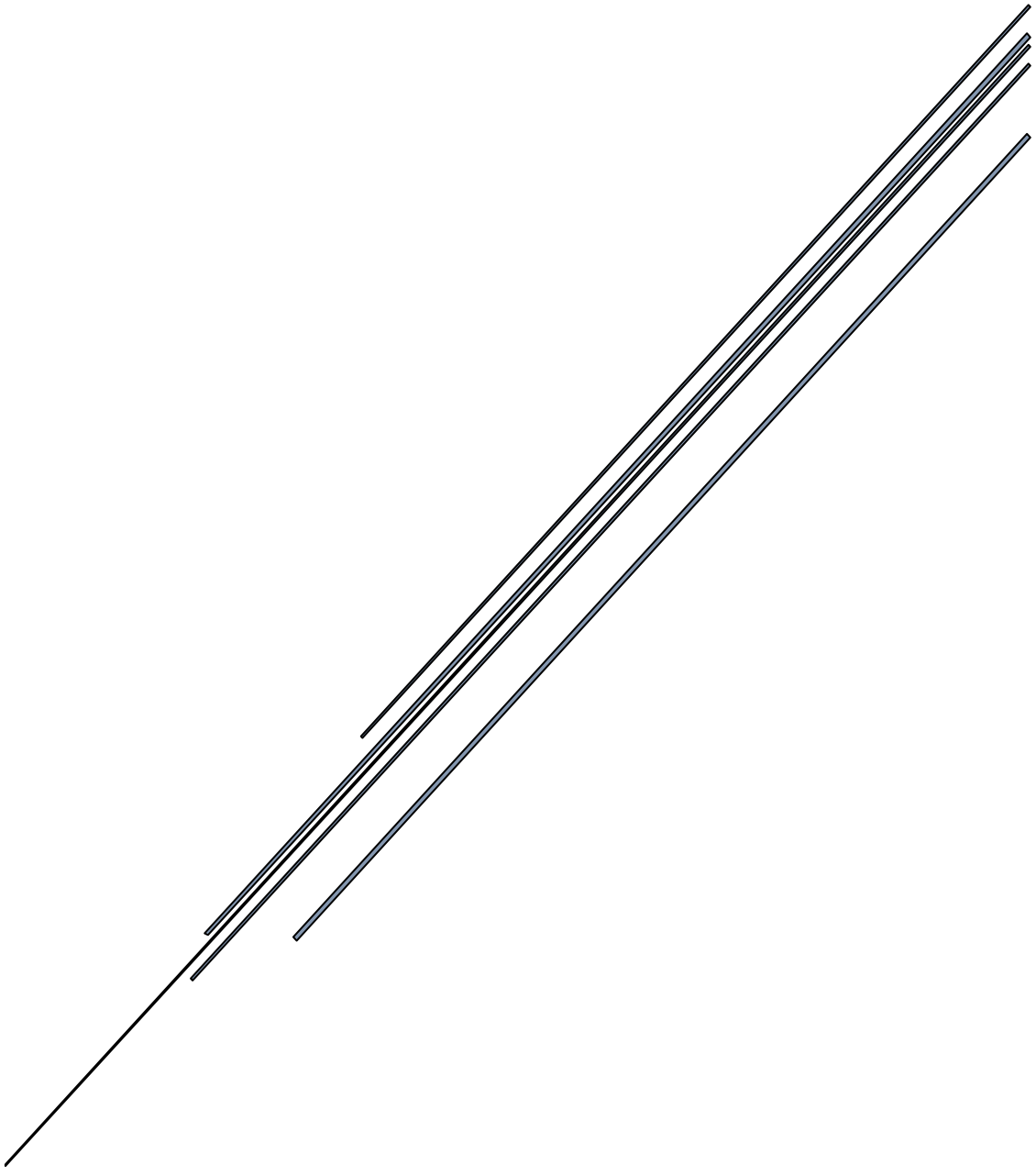
CSEN 901: AI

Project 2 Report



Instructor: Dr. Haythem O. Ismail
Done by: Raneem Wael (43-3851), Abdelrahman Khalifa (43-3878), Heidi Talaat (43-2081), Rowan Bassem (43-4669) – Team 47

Contents

Project Overview

In this project, we are aiming to use Prolog and Notepad to design and implement a logic-based version of the agent we implemented in Project 1 in a simplified world where there is maximum two hostages to save, no pads, no pills, no agents, and the grid's size is either 3×3 or 4×4.

In this version, hostages are not poisoned (no damage increase with time steps). A logic-based agent operates by storing sentences about the world in its knowledge base, using an inference mechanism to infer new sentences, and using these sentences to decide which actions to take.

The design we will be implementing is a form of program where the user can change the input grid size, "Neo's" location, booth location, hostages' locations, and the maximum number of hostages "Neo" can carry in the "KB.pl" file.

They can then run the "Matrix_047.pl" file in Prolog to view the set of actions resulting in a goal state or input a set of actions and see if they result in a goal state.

Design

The project is divided into two main files as follows:

1. "KB.pl": which is a file containing a sample KB that the user can manipulate.
2. "Matrix_047.pl": which is a file containing the implementation.

KB

This file contains a sample input KB:

1. The first line is a predicate representing the size of the grid.
2. The second line is a predicate representing "Neo's" initial location.
3. The third line is a predicate representing the initial locations of the hostages represented as a list of lists of their i and j positions.
4. The fourth line is a predicate representing the location of the telephone booth.
5. The last line is a predicate representing the maximum capacity Neo can carry.

Matrix_047

Initially, this file consults "KB.pl" to be able to use this knowledge base to initialize our matrix elements.

This file contains two main predicates:

1. **goal(S):** this predicate returns a new state if the operator used on the inputted state is valid in accordance with the matrix rules. It generates a plan, using the agent's KB, that "Neo" can follow to collect all the hostages and head to the cell where the telephone booth lies. The result of the query is a situation described as the result of doing some sequence of actions from the initial situation s0.
   - This predicate is divided into two cases:
     a. Case 1: if there are no more hostages to save, we send neo to telephone booth. We do this by first checking that the hostages_loc predicate is empty then call the *state* predicate (explained later) to

ensure that all carried hostages are dropped, then check that "Neo's" location is the same as that of the telephone booth.

b. Case 2: hostages needing to be carried remain. We initialize *S* as "result(drop,_)" as a last action, as when neo drops the final carried hostages, he will be at a goal state. We then call the *state* predicate (explained later). Finally, after the *state* predicate eliminates, we check that "Neo's" location is the same as that of the telephone booth.

2. **state(M,N,X,Y,L,C,R):** this predicate infers a new state using successor state axioms on the current situation. It returns a new state if the operator used on the inputted state is valid in accordance with the matrix rules. It keeps concatenating the actions "Neo" takes to a string result. The *M* and *N* represent the grid width and height, the *X* and *Y* represent "Neo's" location, the *L* represents a list of the locations of hostages left to carry, the *C* represents the number of carried hostages and the *R* is the result string up to the current state.

- For an operator to be valid it needs to follow the following rules:
  a. *Up, down, left, right:* for "Neo" to be able to take any of these actions from the current state, the positions of the new state need to exist within the matrix boundaries. To move *up*; x-1 needs to be greater than or equal to 0, to move *down*; x+1 needs to be less than M, to move *left*; y-1 needs to be greater than or equal to 0, and finally to move *right*; y+1 needs to be less than N. If "Neo" takes any of these actions, his x and y positions are updated for the new state and we concatenate a "result(action,previousResult)" to the beginning of the set of actions outputted.
  b. *Carry:* for "Neo" to be able to carry a hostage he needs to be in the same cell as a hostage remaining not carried or saved (we check for this using the *canCarry* predicate explained later). In addition to that, "Neo" needs to be carrying a number of hostages less than C (the maximum he could simultaneously carry). If a hostage is carried, we remove it from the *hostages_loc* predicate. We also update the current carried number of hostages for the new state and concatenate a "result(carry,previousResult)" to the beginning of the set of actions outputted.
  c. *Drop:* for "Neo" to be able to drop the carried hostages, the number of currently carried hostages needs not be zero and "Neo" needs to be in the same cell as the telephone booth. If "Neo" drops

hostages, his current carrying is set to 0 and we concatenate a "result(drop,previousResult)" to the beginning of the set of actions outputted.

- This predicate is divided into two cases:
  a. Case 1: initializes M, N, X and Y (Neo's location) and the hostages' locations and carried to 0 as initial state s0 (which is concatenated to the result string as "s0").
  b. Case 2: recursively infers a new state using successor state axioms on the current situation.

It contains some other helper functions which are stated below:

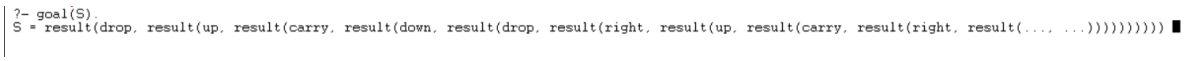
1. **canCarry(X,Y,RemainginHostages,R):** this predicate checks if "Neo" can carry a hostage at the current location. We first check if any remaining hostage exists in current location and if yes, we remove this hostage form the hostages list (using the *remove* predicate which is explained later). The *X* and *Y* represent "Neo's" location and *R* is the result string up to the current state.
2. **remove(X,Y,[H|T],T):** this predicate removes a pair representing a location of X and Y from a list of locations.
   - This predicate is divided into two cases:
     a. Case 1: base case – we check if the position is the same as the first location in the list. The *H* here is represented as an [X,Y] list.
     b. Case 2: recurse through the list to find the location by diving the head into another head and tail then passing the new tail and end tail to the *remove* predicate again.

Examples

We used the example queries already present in the project description to show that our implementation works.

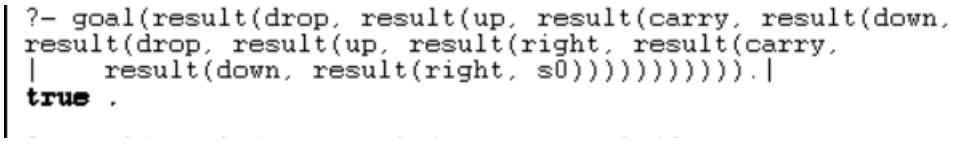
Example Query 1

goal(S).



Example Query 2

goal(result(drop, result(up, result(carry, result(down, result(drop, result(up, result(right, result(carry, result(down, result(right, s0)))))))))).



Example Query 3

goal(result(up, result(carry, result(down, result(drop, result(up, result(right, result(carry, result(down, result(right, s0)))))))))).