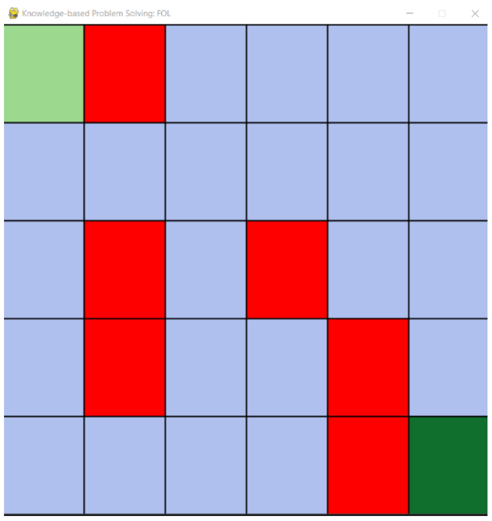
Artificial Intelligence for Big Data Systems

Assignment 2: Knowledge-Based Problem Solving

(Program Report)

Stefan Velev, 0MI3400521

We are given the problem of finding a path between the starting node (0,0) and the end node (4,5) of a maze. The permitted moves are four: Left, Right, Up and Down. The red nodes contain obstacles and no movements are possible through them. The boundaries are walls which are blocking the movements.



We have to find the path from the starting node to the end node of the maze using knowledge base containing that representation of the knowledge and applying logical inference. We will do that with the help of the ***Prolog*** logic language that I will use for finding the path, and ***Python*** being used for visualisation. The programming logic is the following – we start with the ***Python*** program which represents the grid in the same way as it was in coursework 1. After that, we use the ***PySwip*** library to make a query to the ***Prolog*** file that contains the main pathfinding logic (facts, rules) of the maze.

The details regarding the maze board are the same as in my solution to coursework 1 so we will not pay attention to them again. The main difference is that we do not execute any search algorithm in the ***Python*** program. With the help of the ***PySwip*** library we turn to the ***Prolog*** program which returns the found paths. After that, we visualise them as an animation step by step in the ***Python*** program.

```python
def colourise(draw, grid):
    prolog = Prolog()
    prolog.consult(PROLOG_FILE_NAME)
    result_list = list(prolog.query(PROLOG_QUERY))[0]

    for current_cell in result_list['Path']:
        grid[int(current_cell[2])][int(current_cell[1])].make_solution_path()
        draw()
        pygame.display.update()
        pygame.time.delay(ANIMATION_TIME_MILLISECONDS)

    pygame.display.set_caption('Solution found with FOL')
    return True
```

There are two pairs of programs in ***Python*** and ***Prolog*** (cw2-bc-Stefan-Velev.py/pl & cw2-optimized-Stefan-Velev.py/pl) – one without optimisation and one optimised. We will start explaining them in detail.

In the first ***Prolog*** program we start with facts about the maze. The main facts that I decided to use for the solution are direct paths between neighbouring nodes. In that case, I do not need separate facts for representing the obstacles (walls). Each fact relies on the predicate direct_path/2 to indicate a direct movement from cell X to cell Y (X and Y are neighbouring cells).

```
direct_path(c00, c10).
direct_path(c10, c11).
direct_path(c10, c20).
direct_path(c20, c30).
direct_path(c30, c40).
direct_path(c40, c41).
direct_path(c41, c42).
direct_path(c42, c43).
direct_path(c42, c32).
direct_path(c43, c33).
direct_path(c33, c32).
direct_path(c32, c22).
direct_path(c22, c12).
direct_path(c11, c12).
direct_path(c12, c02).
direct_path(c12, c13).
direct_path(c02, c03).
direct_path(c13, c03).
direct_path(c13, c14).
direct_path(c03, c04).
direct_path(c04, c05).
direct_path(c04, c14).
direct_path(c14, c15).
direct_path(c14, c24).
direct_path(c24, c25).
direct_path(c05, c15).
direct_path(c15, c25).
direct_path(c25, c35).
direct_path(c35, c45).
```

We continue with the rules. We define a move between cells as a direct path in both directions regardless how it was previously defined. The move/2 rule abstracts bidirectional movement so the program can check if two cells are directly connected regardless of direction.

```
move(X, Y) :- direct_path(X, Y).
move(X, Y) :- direct_path(Y, X).
```

We continue with the core rules of the program. These are the rules connected to inter_path/4. It is used for recursively constructing the paths from the starting cell (S) to the final cell (F). As a third argument it accepts a list which is used for storing all the visited nodes. It is required since otherwise we would not be able to guarantee that we will not make redundant moves resulting in a never-ending program. The fourth argument is a list which we use for storing the result to be returned at the end.

```
inter_path(S, F, _, [S, F]) :-
    move(S, F).

inter_path(S, F, Visit, [S | Rest]) :-
    move(S, Z),
    Z \= F,
    not(member(Z, Visit)),
    inter_path(Z, F, [S | Visit], Rest).
```

With the path/1 predicate we set the program specifying the starting node (c00), the end node (c45), the visited list (initially it is empty – []) and the only argument of the path/1 to be returned with the solution path(s). With the help of the built-in write/1 predicate, I print the entire result in the ***Python*** terminal in addition to the visualisation which can be seen in the separate window. If we just prefer the written output without any result to be returned, we can make a query to path/0 for which the returned path is anonymous and therefore not returned.

```
path() :- path(_).

path(Path) :-
    inter_path(c00, c45, [], Path),
    write(Path), nl.
```

We continue with the second pair of programs which represents the optimised version of the path. The main difference consists of the fact that in this case we print the shortest path (if more than one, we pick the first), whereas in the first version of the ***Prolog*** program we visualised the first found path (regardless of its length). This is achieved with the same facts. The rules for move/2 are the same. The logic for the inter_path/5 predicate is similar but more details need to be added for the path length and specifically for storing the length of the shortest path.

```
inter_path(S, F, Visit, [S, F], PathLength) :-
    move(S, F),
    PathLengthIncreased is PathLength + 1,
    nb_getval(globalMinPath, CurrentMinPath),
    PathLengthIncreased =< CurrentMinPath,
    nb_setval(globalMinPath, PathLengthIncreased).
```

With the above rule (base case) we say that if S and F are directly connected (move(S, F)), then we increment the current path length, check if this new path length is less than or equal to the current shortest path (globalMinPath) and if it is, we update the globalMinPath to this new shorter path length.

```
inter_path(S, F, Visit, [S | Rest], PathLength) :-
    move(S, Z),
    Z \= F,
    not(member(Z, Visit)),
    PathLengthIncreased is PathLength + 1,
    nb_getval(globalMinPath, CurrentMinPath),
    PathLengthIncreased =< CurrentMinPath,
    inter_path(Z, F, [S | Visit], Rest, PathLengthIncreased).
```

With the above rule (recursive case) we recursively explore the maze in the following order – from the current node S we move to a neighbouring node Z, ensure that Z is not already the end node and it is not in the visited list to avoid cycles, increment the path length and check if it is less than or equal to the current shortest path and if it is we continue exploring paths from Z by adding S to the visited list, removing S for the explored (Rest) list and using the newly calculated path length. In that way, we build paths recursively while tracking and pruning those of them that exceed the current shortest path length.

```
find_path_of_length(PathList, PathLength, OptimalPath) :-
    member(A, PathList),
    length(A, CurrentPathLength),
    CurrentPathLength = PathLength,
    OptimalPath = A.
```

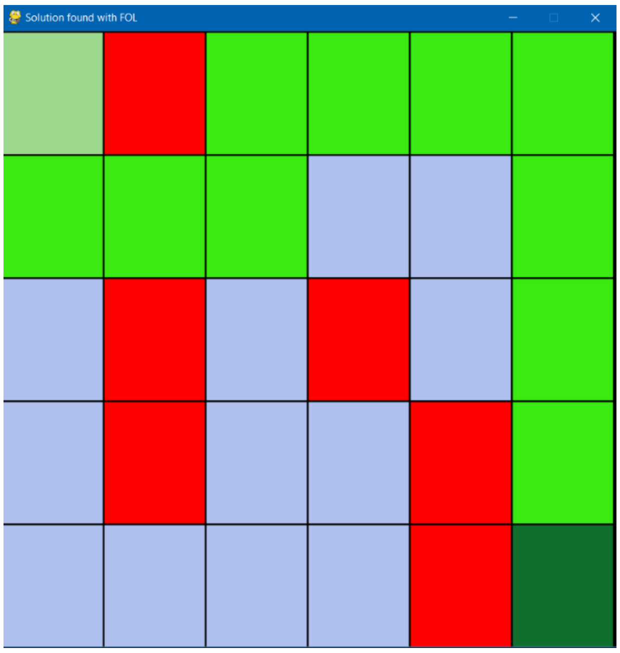
With the find_path_of_length/3 rule, we select a path from a list of paths (PathList) that matches the given PathLength. The result is stored in the OptimalPath argument. First, we get an element from the list of paths. We store the current path length in CurrentPathLength using the built-in ***Prolog*** predicate length/2 that calculates the number of elements in a list. The next step is to compare the calculated path length (CurrentPathLength) with the desired length (PathLength). If they are equal, we continue to the next (final) clause which binds the current path (A) to the OptimalPath variable. Otherwise, ***Prolog*** backtracks and tries with the next element of the provided list.

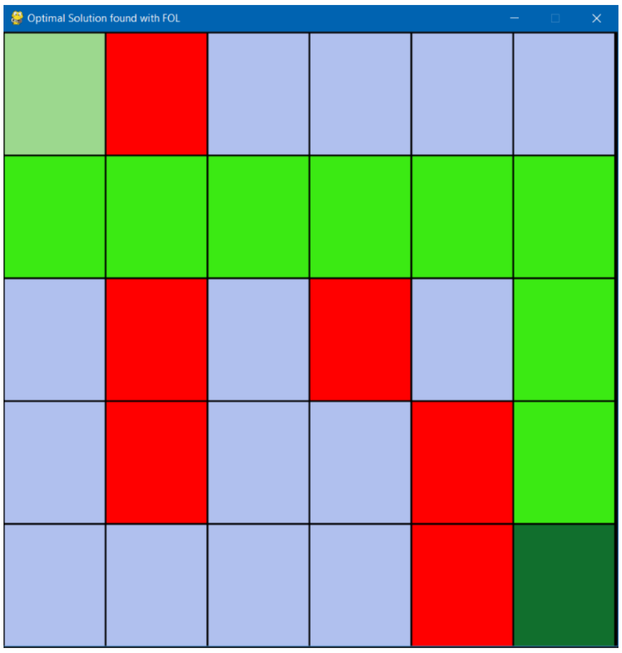
```
optimal_path(Visit, Path, PathLength) :-
    nb_setval(globalMinPath, 1000),
    PathLength = 1,
    findall(Path, (Visit = [], inter_path(c00, c45, Visit, Path, PathLength)), PathList),
    nb_getval(globalMinPath, PrintPath),
    find_path_of_length(PathList, PrintPath, OptimalPath),
    write(OptimalPath), nl.

optimal_path(Path) :-
    optimal_path([], Path, _).
```

The optimal_path/3 predicate is the main predicate responsible for finding the solution. It finds and prints a single shortest path from the starting node (c00) to the end node (c45). It does so by iterating through all possible paths and filtering the one with the shortest length. In the first subclause a global variable is initialized to a high value (1000) as the start of the computation. This variable acts as a threshold for the maximum allowed path length. It ensures that longer paths are pruned early in the search. After that, we set the base value of PathLength to 1 since the actual path length will increase dynamically as the search progresses. The next subclause relies on the findall/3 built-in ***Prolog*** predicate that generates all possible solutions for a given query and collects them into a list. Path is the variable that represents each valid path from c00 to c45. PathList is the resulting list of all paths generated by the query. With nb_getval(globalMinPath, PrintPath) we retrieve the current value of globalMinPath (updated with the minimal length by the previous inter_path/5 rules). The next step is to use the find_path_of_length/3 rule in order to find a path that matches the shortest length. Finally, we print the resulting optimal path followed by a new line (nl).

The results from both the ***Python*** programs can be seen below:


Solution found with FOL


Optimal Solution found with FOL