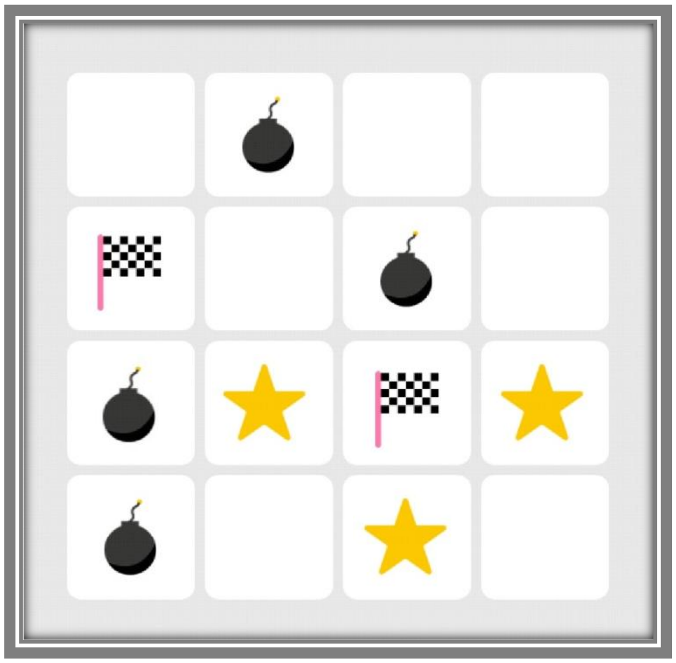
Cairo University
Faculty of Computers & Artificial Intelligence
Department of Computer Science

Artificial Intelligence Course
Assignment 1

1- Path to Safety (Game): *[2 marks]*

❖ *About the game:*

Given an MxN board, the goal of the game is to move from the 'Start' square and safely reach the 'Finish' square. In order to "safely" reach the finish square, you can't pass through any squares containing bombs. You should also try to collect as many stars as possible in your path. The game has several levels; each level is different in configuration and some levels have more than one path. For example, the level shown in the opposite figure has **3** possible safe paths. The shortest path contains 1 star while the longest contains 3.



❖ *What you are required to do:*

You will be given a prolog source file containing the level data (like the file attached with this document) and you are required to make a prolog program that produces a list of moves for each possible safe path from the start to the finish square and states the number of stars that can be collected in each path. Your prolog source file should only contain the predicates and rules needed by the solver; however, it shouldn't define any facts about the level configuration.

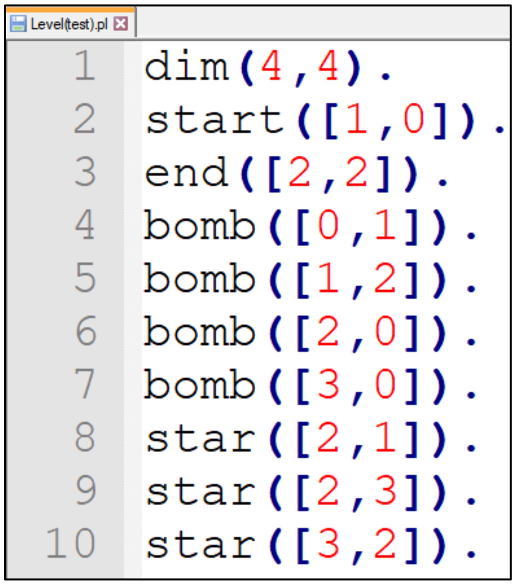
❖ *A sample test case:*

This test case shows the sample input that should be entered and what the output should look like. Notice that this output is the output produced when we run the solver on the level shown in the first figure.

```
?- load_files("C:/ Level3.pl"). %the level file (test case) given by your TA
true
?- load_files("C:/ Solver.pl"). %the file containing your program
true
?- play(Moves,Stars). %the user will only enter this predicate (play/2)
Moves = [right, down, down, right, up],
Stars = 2 ;
Moves = [right, down, down, right, right, up, left],
Stars = 3 ;
Moves = [right, down, right],
Stars = 1 ;
false.
```

❖ *The structure of the input file:*

You will find the level file (for the level in the first figure) attached with this document. All level files will have the same structure shown in the opposite figure. A level file contains the following predicates:



```
dim(4,4).
start([1,0]).
end([2,2]).
bomb([0,1]).
bomb([1,2]).
bomb([2,0]).
bomb([3,0]).
star([2,1]).
star([2,3]).
star([3,2]).
```

- **dim/2:** the dimensions of the board.
- **start/1:** the start square.
- **end/1:** the finish square.
- **bomb/1:** the square that contains a bomb.
- **start/1:** the square that contains a star.

You will notice that each square is represented by a list containing 2 elements; the index of the row and the index of the column in which this square is located. Level files will have the **same structure** (same predicate names, number of arguments), however they will have **different data** (different dimensions, start/end locations, different numbers and locations of bombs and stars). **You are not allowed to change the structure of the level file**; you should adjust your program to work with this structure.

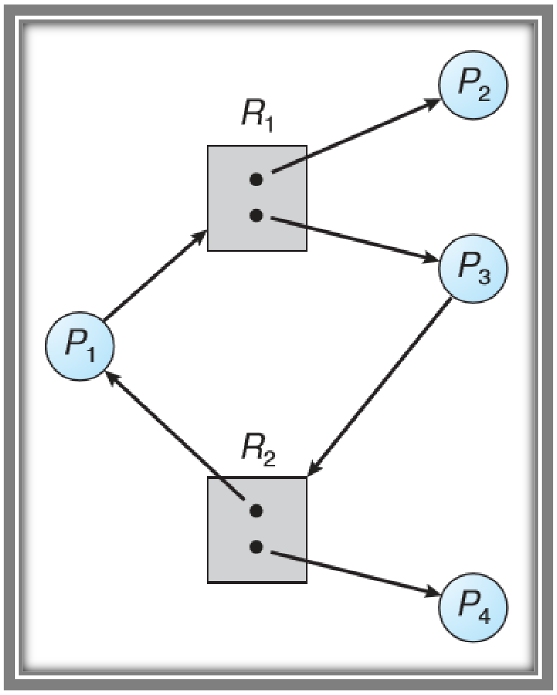
❖ *Important Notes: (Please read these notes carefully to avoid losing grades)*

- The user will interact with your program through **the predicate "play" which takes exactly 2 arguments**; a variable for the list of moves and another for the number of stars.
- Various level files will be used to test your program, so **don't put the level facts (dim, start, …)** in your prolog source file.
- Make sure your program **handles cycles** (i.e. don't let your program get stuck by returning to the same square repeatedly).
- The output of your program should look like the **output in the sample test case**.
- For this problem, **don't use built-in predicates** like findall, aggregate_all, foreach, …

2- Resource-Allocation Graph: *[4 marks]*

❖ *About the graph:*

The graph consists of a set of vertices and a set of edges. The vertices represent all the active processes in the system {P_1, P_2, ..., P_n} as well as all the resource types in the system {R_1, R_2, ..., R_m}. The dots found in each resource square represent the number of instances of that resource. A directed edge from process P_i to resource type R_j ($P_i \rightarrow R_j$) signifies that Pi has requested an instance of R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i ($R_j \rightarrow P_i$) signifies that an instance of R_j has been allocated to P_i.



A process releases its allocated resources only when it acquires all the resources it needs (all its requests are fulfilled). So, if the graph contains no cycles, no process in the system is deadlocked. Otherwise, a deadlock may exist. If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. On the other hand, if each resource type has several instances, a cycle doesn't necessarily imply that a deadlock has occurred. In the previous figure, a cycle exists; however, there is no deadlock. This is because process P_4 can release its instance of resource R_2. That resource can then be allocated to P_3, breaking the cycle.

❖ *What you are required to do:*

You are required to make a prolog program that determines whether a graph can reach a safe state (with no deadlock) or not and if the graph can reach a safe state, you should output the process execution order that made the graph be in such a state. You will have to define the graph data as facts and define the necessary predicates/rules that detect deadlocks. **For simplicity, assume we only have 2 resource types (R_1 and R_2).**

❖ *A sample test case:*

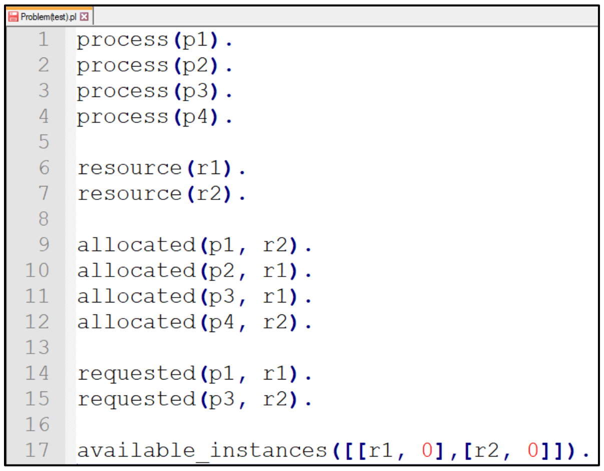
This test case shows the sample input that should be entered and the sample output for the graph in the previous figure.

```
?- safe_state(X).
X = [p2, p4, p1, p3].
```

Notice that that graph has different solutions.

❖ *The input file:*

The input file should look somewhat like the opposite figure, but you can adjust the facts/predicates if you need to.



```
process(p1).
process(p2).
process(p3).
process(p4).

resource(r1).
resource(r2).

allocated(p1, r2).
allocated(p2, r1).
allocated(p3, r1).
allocated(p4, r2).

requested(p1, r1).
requested(p3, r2).

available_instances([[r1, 0],[r2, 0]]).
```

Important Notes: *(Please read these notes carefully to avoid losing grades)*

- In this assignment, you will work in teams. **The minimum number of students in a team is 2 and the maximum is 3.**
- Please submit **one compressed folder** containing the **2 (.pl) files**. The folder name should follow this structure: **ID1_ID2_ID3_GX,Y**
- **Cheating students will take -6** and no excuses will be accepted. If you have any problems during the submission, contact your TA but don't, under any circumstances, give your code to your friends.

Grading Criteria:

Problem 1	
Suitable predicates & rules	**1**
Output (moves list & number of stars)	**1**
Problem 2	
Suitable representation of graph info	**0.5**
Suitable predicates & rules	**2**
Output (true + execution order / false)	**1.5**
Bonus: Solve the problem for graphs with any number of resources.	**1**
Total ***(6 + 1 bonus grade)***	**7**

Good luck