U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO
DEPARTMENT OF INFORMATICS ENGINEERING

U.PORTO FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO
DEPARTMENT OF COMPUTER SCIENCE

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2021/2022 – 1st Semester

Prolog

Database Modification, Graphs and Search

Daniel Castro Silva
dcs@fe.up.pt

Agenda

- Database Modification
    - Memoization
- Cycles
- Graphs and Search
    - Puzzles and Games

Database Modification

- Prolog allows clauses to be dynamically added or removed from a program
  - This provides great flexibility
  - However, modifying the program is costly, as it requires re-indexing

- In order to add or remove clauses from a predicate, it first needs to be declared dynamic

```
:-dynamic male/1, female/1, parent/2.
```

See section 4.12 of the SICStus Manual for more information

Adding Clauses

- ***assert/1*** adds a new clause to the program; there are two additional variations of this predicate:
  - ***asserta/1*** – the new clause is added before all existing predicate clauses (if any)
  - ***assertz/1*** – the new clause is added after all existing predicate clauses (if any)

```
ask_and_add_to_kb:-
      write('Insert Parent-Child to add'),nl,
      read(P-C),
      assert(parent(P, C)).
```

When adding a rule, an additional pair of parentheses is required

Removing Clauses

- ***retract/1*** removes a clause from the program (the first that matches the given clause)
- ***retractall/1*** retracts all clauses matching the specified head
- ***abolish/1*** removes all clauses and properties of the specified predicate

```
retractall(parent(homer, _)).
abolish(parent/2).
```

```
replace_definition:-
      retract(( ancestor(X,Y):-parent(X,Y) )),
      asserta(( ancestor(X,Y):-father(X,Y) )),
      asserta(( ancestor(X,Y):-mother(X,Y) )).
```

Predicate Listing

- ***listing/0*** lists all clauses from the currently loaded program
- ***listing/1*** lists all clauses from a given predicate

- These predicates list the code in the current output stream
  - Note that variable naming and code formatting are not preserved

```
a(X, Y):- b(X), !, b(Y).
a(3, 4).
b(2).
b(3).
```

```
| ?- listing.
a(A, B) :-
        b(A), !,
        b(B).
a(3, 4).

b(2).
b(3).

yes
| ?- listing(a/2).
a(A, B) :-
        b(A), !,
        b(B).
a(3, 4).

yes
```

Accessing Clauses

- ***clause(+Head, ?Body)*** allows access to the clauses of a given predicate in the knowledge base

```
a(X, Y):- b(X), !, b(Y).
a(3, 4).
b(2).
b(3).
```

```
| ?- clause( a(X,Y), _Body ),
     retract(( a(X,Y):- _Body )),
     a(A, B),
     asserta(( a(X,Y):- _Body )).
A = 3,
B = 4 ?
yes
| ?- listing(a/2).
a(A, B) :-
        b(A), !,
        b(B).
a(3, 4).

yes
| ?- clause( a(X,Y), Body ), retract(( a(X,Y):-Body )).
Body = (b(X),!,b(Y)) ?
yes
| ?- listing(a/2).
a(3, 4).

yes
```

Database Modification

- Assert and retract should be used sparingly (ideally only for things that do not change often)
  - They are slow operations
  - It can make programs harder to understand / debug

- The effect of database modification predicates is not undone in backtracking (just like input/output)

Memoization

- Modifying the database can be used to save partial results, resulting in a dynamic programming approach

```
fib(0, 0).
fib(1, 1).
fib(N, F):- N > 1,
      N2 is N-2, N1 is N-1,
      fib(N2,F2), fib(N1,F1),
      F is F2 + F1.
```

```
fib(0, 0).
fib(1, 1).
fib(N, F):- N > 1,
      N2 is N-2, N1 is N-1,
      fib(N2,F2), fib(N1, F1),
      F is F2 + F1,
      asserta(( fib(N,F):-! )).
```

Could we use *assertz* instead?

Database Modification

- We can also use this approach as an alternative to finding all answers to a query

```
get_all_children(Parent, _Children):-
      assert( children(Parent, []) ),
      fail.
get_all_children(Parent, _Children):-
      parent(Parent, Child),
      retract( children(Parent, Current) ),
      assert( children(Parent, [Child|Current]) ),
      fail.
get_all_children(Parent, Children):-
      retract( children(Parent, Children) ).
```

Why is this approach inefficient?

Failure Driven Loops

- The example above is a failure driven loop
  - The *fail* forces Prolog to backtrack until all solutions are found

```
failure_driven_loop:-
      find_solution(X),
      do_something_with_solution(X),
      fail.
failure_driven_loop.          %ensure predicate succeeds
```

  - Efficient in terms of memory use
  - Usually only used in situations when only side effects are important (results are not kept)

Failure Driven Loops

- Failure driven loops are an alternative to recursive ones
  - Compare the following two approaches to implement a predicate *print_n(N, C)*, which prints a character *C* to the console *N* times

```
print_n(0, _C):- !.
print_n(N, C):-
      write(C),
      N1 is N-1,
      print_n(N1, C).
```

```
print_n(N, C):-
      between(1, N, _T),
      write(C),
      fail.
print_n(_N, _C).
```

Which approach is more efficient?

Failure Driven Loops

- Another example: consulting a program

```
consult(File):-
      see(File),
      loop,
      seen.

loop:-
      repeat,
      read(Clause),
      process(Clause), !.

process(end_of_file):- !.
process(Clause):-
      assert(Clause),
      fail.
```

Generic Game Program

- A generic game program can be coded with a recursive loop

```
play_game:-
      initial_state(GameState-Player),
      display_game(GameState-Player),
      game_cycle(GameState-Player).

game_cycle(GameState-Player):-
      game_over(GameState, Winner), !,
      congratulate(Winner).
game_cycle(GameState-Player):-
      choose_move(GameState, Player, Move),
      move(GameState, Move, NewGameState),
      next_player(Player, NextPlayer),
      display_game(GameState-NextPlayer), !,
      game_cycle(NewGameState-NextPlayer).
```

Generic Game Program

```
choose_move(GameState, human, Move):-
      % interaction to select move
choose_move(GameState, computer-Level, Move):-
      valid_moves(GameState, Moves),
      choose_move(Level, GameState, Moves, Move).

valid_moves(GameState, Moves):-
      findall(Move, move(GameState, Move, NewState), Moves).

choose_move(1, _GameState, Moves, Move):-
      random_select(Move, Moves, _Rest).
choose_move(2, GameState, Moves, Move):-
      setof(Value-Mv, NewState^( member(Mv, Moves),
              move(GameState, Mv, NewState),
              evaluate_board(NewState, Value) ), [_V-Move|_]).

% evaluate_board assumes lower value is better
```

Graphs and Search

- Graphs can be represented as the connections between nodes
  - set of facts representing [directed] edges

```
connected(porto, lisbon).
connected(lisbon, madrid).
connected(lisbon, paris).
connected(lisbon, porto).
connected(madrid, paris).
connected(madrid, lisbon).
connected(paris, madrid).
connected(paris, lisbon).
```

Depth-First Search

- Searching for a possible connection between nodes is made easy by Prolog's standard depth-first search mechanism

```
connected(porto, lisbon).
connected(lisbon, madrid).
connected(lisbon, paris).
connected(lisbon, porto).
connected(madrid, paris).
connected(madrid, lisbon).
connected(paris, madrid).
connected(paris, lisbon).
```

```
connects_dfs(S, F):-
      connected(S, F).
connects_dfs(S, F):-
      connected(S, N),
      connects_dfs(N, F).
```

```
| ?- connects_dfs(porto, madrid).
yes
| ?- connects_dfs(madrid, porto).
```

When does this approach fail?

Depth-First Search

- Adapted solution with an accumulator to avoid loops

```
connected(porto, lisbon).
connected(lisbon, madrid).
connected(lisbon, paris).
connected(lisbon, porto).
connected(madrid, paris).
connected(madrid, lisbon).
connected(paris, madrid).
connected(paris, lisbon).
```

```
connects_dfs(S, F):-
      connects_dfs(S, F, [S]).

connects_dfs(F, F, _Path).
connects_dfs(S, F, T):-
      connected(S, N),
      not( member(N, T) ),
      connects_dfs(N, F, [N|T]).
```

What would we have to change to *return* the connecting path (route)?

Breadth-First Search

- We can also easily create a BFS solution using *findall*

```
connected(porto, lisbon).
connected(lisbon, madrid).
connected(lisbon, paris).
connected(lisbon, porto).
connected(madrid, paris).
connected(madrid, lisbon).
connected(paris, madrid).
connected(paris, lisbon).
```

```
connects_bfs(S, F):-
      connects_bfs([S], F, [S]).

connects_bfs([F|_], F, _V).
connects_bfs([S|R], F, V):-
      findall(N,
      ( connected(S, N),
        not(member(N, V)),
        not(member(N, [S|R]))), L),
      append(R, L, NR),
      connects_bfs(NR, F, [S|V]).
```

What would we have to change to *return* the connecting path (route)?

Games and Puzzles

- Prolog (and search) can easily be used to search for a solution to one-person games or puzzles
- States represented as the nodes of the graph
  - Initial state is the starting node
  - Winning conditions define the final nodes
- Movements represented as the transitions between nodes
  - States don’t need to be represented in extension – transitions can specify new states based on the previous one and the move made

Generic Solver

- A generic [abstract] solver to one-person games/puzzles

```
initial(InitialState).

final(State):- winning_condition(State).

move(OldState, NewState):- valid_move(OldState, NewState).

play:-   initial(Init),
         play(Init, [Init], States),
         reverse(States, Path), write(Path).

play(Curr, Path, Path):- final(Curr), !.
play(Curr, Path, States):-    move(Curr, Next),
                              not( member(Next, Path) ),
                              play(Next, [Next|Path], States).
```

Games and Puzzles

- Example: fill a 5-gallon jug with 4 gallons of water, using the 5-gallon jug and a 3-gallon jug

```
initial(0-0).    % Jug5-Jug3

final(4-_).

move(_-S, 5-S). % fill jug 1
move(F-_, F-3). % fill jug 2
move(_-S, 0-S). % empty jug 1
move(F-_, F-0). % empty jug 2
move(F-S, NF-NS):- NF is max(0, F-(3-S)), NS is min(3, F+S). % 1->2
move(F-S, NF-NS):- NF is min(5, F+S), NS is max(0, S-(5-F)). % 2->1
```

Shortest Path

- To find the smallest set of plays we just need to find all paths and select the shortest one
  - Easily accomplished using *setof*

Is DFS the best way of doing this?

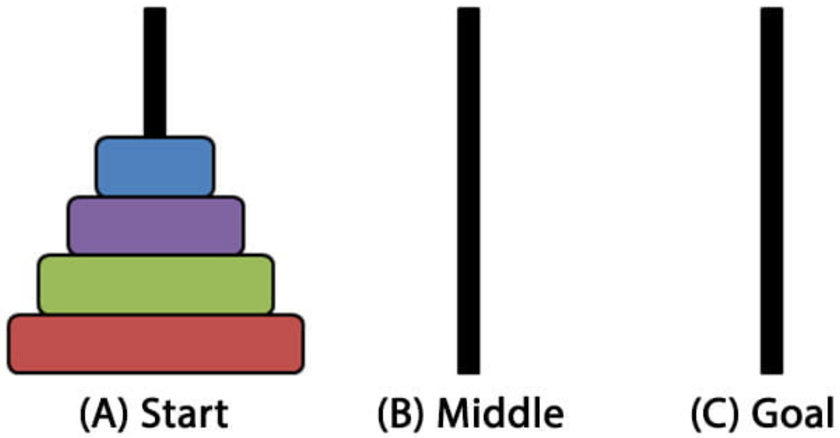
```
play:-   initial(Init),
         setof( Length-Path, (
                     play(Init, [Init], Path),
                     length(Path, Length) ),
              [_ShortestLength-States|_] ),
         reverse(States, Path), write(Path).
```

What if we wanted the path with the lowest cost?

How could we obtain all paths with shortest length?

Games and Memoization

- Example: Tower of Hanoi
  - Goal: move stack from pole 1 to pole 3
  - Rules:
    - Can only move one disk at a time
    - Disks can only be placed on top of a larger disk

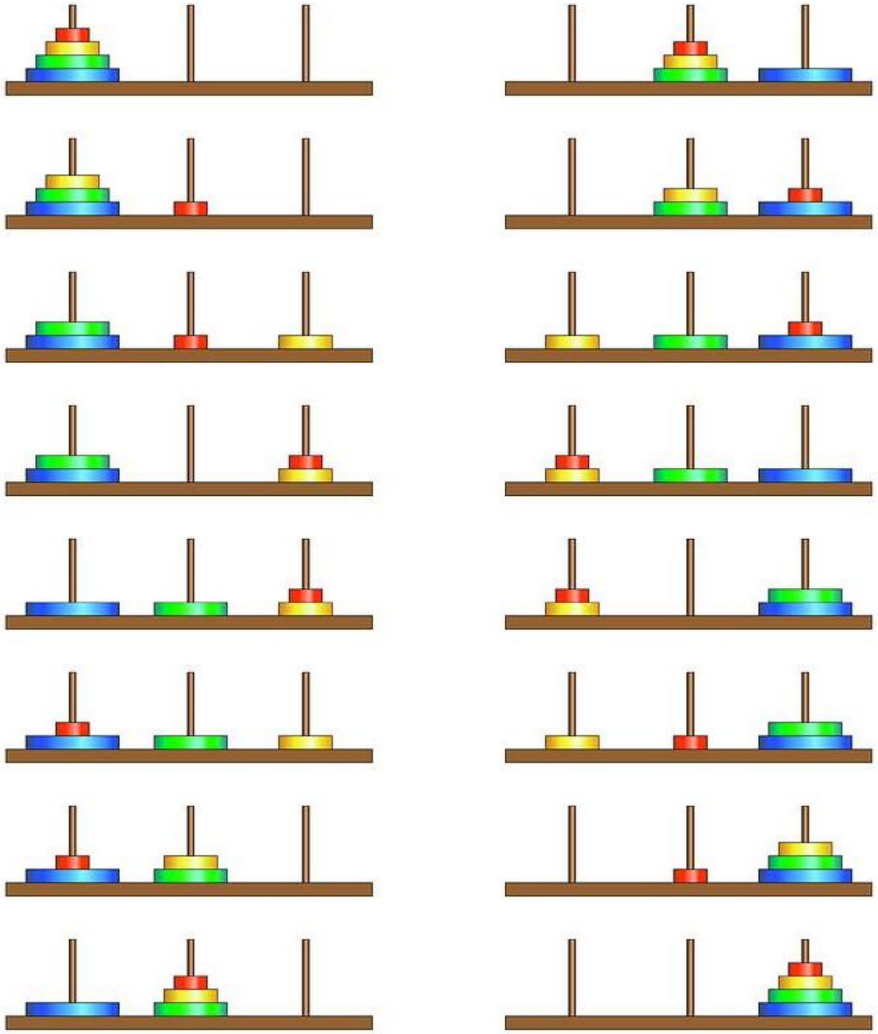


Games and Memoization

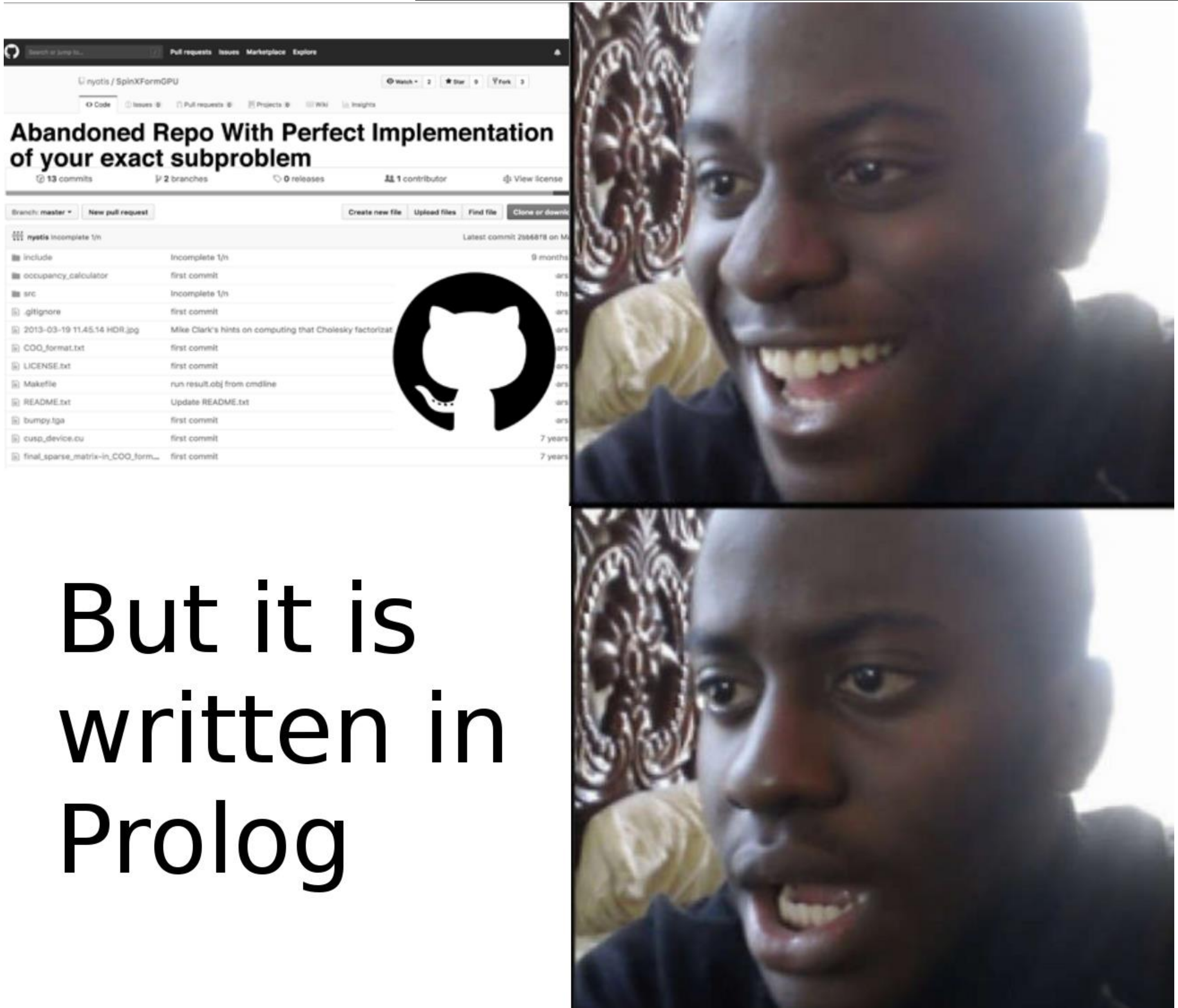
- To move a stack of size N from pole 1 to pole 3, first move stack of size N-1 to pole 2, move base piece, and then move N-1 stack from pole 2 to pole 3

```
hanoi(1, A, B, C, [A-C]).
hanoi(N, A, B, C, Moves):-
      N > 0, N1 is N-1,
      hanoi(N1, A, C, B, Fst),
      hanoi(N1, B, A, C, Lst),
      append(Fst, [A-C|Lst], Moves),
      asserta(hanoi(N, A, B, C, Moves)).

test_hanoi(N, A, B, C, M):-
      hanoi(N, X, Y, Z, M), X-Y-Z=A-B-C.
```



Q & A


Abandoned Repo With Perfect Implementation of your exact subproblem
But it is written in Prolog